

Multimedia Retrieval

Chapter 4: Image Retrieval

Dr. Roger Weber, roger.weber@gmail.com

[4.1 Introduction](#)

[4.2 Visual Perception](#)

[4.3 Image Normalization](#)

[4.4 Image Segmentation](#)

[4.5 Color Information](#)

[4.6 Texture Information](#)

[4.7 Shape Information](#)

[4.8 Blob Recognition \(unsupervised clustering\)](#)

[4.9 Simple Neural Network Classifier](#)

[4.10 Deep Learning](#)

[4.11 Literature and Links](#)

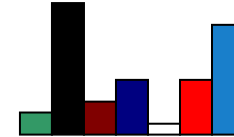
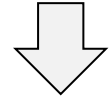


4.1 Introduction

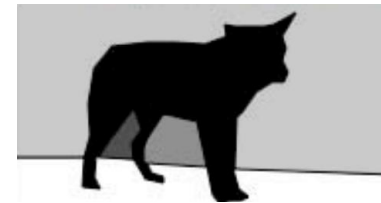
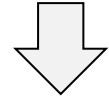
- We already talked about the semantic gap previously:
 - With multimedia content, the raw material (signal information, pixels) is not suitable for query matches. For example, the wolf on the right hand side is a set of thousands of pixels that are interpreted by our brain as a depiction of an animal. But there is no straightforward correlation between the pixels and the concept of animal. This is the so-called semantic gap, i.e., we can not ask with natural language and match that directly to the signal information.
 - To close the semantic gap, we need to extract concepts from the signal information and bring it to a level that allows users to match their information need
- In the following, we start with image data:
 - First, we have a closer look at human perception (color, form, shape) and describe perception with low-level feature descriptors (e.g., color distribution). With similarity search, we can bridge the semantic gap
 - Second, we use learning approaches to extract concepts and classify the content in various ways. These classifiers can be treated like meta data or text annotations. In a later chapter, we also combine similarity and text/meta data based search methods



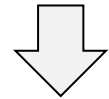
Raw Media



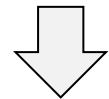
Descriptors



Objects
(segmentation)



Object Labels
(segmentation)



Wolf on Road with Snow on
Roadside in Yosemite
National Park, California on
Jan 24, 2004

Semantics

- Feature design for images



- **Image Normalization** includes a number of pre-processing steps including noise elimination, normalization of signal information, adjustments and corrections of the raw data. For example, when analyzing frames in an interlaced video sequence, deinterlacing is a typical step to reduce combing effects that interfere with feature extraction. Heavily depends on the data set.
- **Image Segmentation** partitions the image into sub-areas for which perceptual features are extracted. We distinguish between global features (for the entire image) and local features (for a region within the images). If we have local features, the aggregation step (4) is necessary to obtain a global feature for the image.
- **Feature Extraction** describes the signal information based on perceptual aspects such as color, texture, shape, and points of interest. For each category, a number of methods exists with different invariances (e.g., robustness against scaling, translation, rotation). We do not consider labeling of images in this chapter (see the next chapter for high-level features)
- **Feature Aggregation** summarizes perceptual features to construct a final descriptor (or a set of descriptors). The aggregation often uses statistical approaches like mean values, variances, covariances, histograms, and distribution functions. With local features, we can further derive statistical measure across the regions (e.g., self-similarity, mean values, variances, covariances). In the following we often discuss feature aggregation together with the feature extraction method.

- The definition of similarity also comes with mapping to invariances, i.e., changes applied to the material that do not impact similarity (or only have a small impact). Examples include:
 - Translation invariant: (small) shifts of the picture have no significant impact on feature values
 - Rotation invariant: rotations of the image have no significant impact on feature values
 - Scale invariant: up- or down-sampling does not change the feature value. Note that scale differences are very common due to different image resolutions. In the absence of a normal sized scale, it is even more important to demand scale invariance
 - Lightning invariant: Adjustments of lightning (daylight, artificial light, brightness adjustments, gamma corrections) have no significant impact on feature values
 - Noise robustness: noise, JPEG artefacts, quantization errors, or limited color gamut have no significant impact on feature values
- Invariances are important to recognize the same objects under different conditions. For instance, Shazam is presented with recordings of “bad quality” due to background noise, audio recording issues (for instance, you are recording in a bar with poor loudspeakers), or people talking over the music. The features used by Shazam must be robust enough to be invariant for a wide range of alterations of the raw signal information (user is not able to prevent a “perfect sample”). This goes much further than just spelling corrections in text retrieval. The design of such features is beyond the material of this course, but we look at some of the basic aspects of perception and invariance.

- A very common method to measure similarity is through a distance function. Assume we have a feature space \mathbb{R}^d with d dimensions. A query Q is mapped into this feature space yielding a feature vector $\mathbf{q} \in \mathbb{R}^d$. The same mapping leads to feature vectors $\mathbf{p}_i \in \mathbb{R}^d$ for each of the media objects P_i . In case of uncorrelated dimensions, a weighted L_k -norm is a good selection to measure distances
 - The weights are chosen such that the ranges of all dimensions become comparable. Several strategies exist to compute the weights. Here are two examples:

$$w_j = \frac{1}{\max_i p_{i,j} - \min_i p_{i,j}}$$

$$w_j = \frac{1}{\sigma_j} \quad \text{with } \sigma_j \text{ being the standard deviation of values in dimension } j$$

- The distance between the query vector \mathbf{q} and media vector \mathbf{p}_i is then:

- L_1 -norm or Manhattan distance:

$$\delta(\mathbf{q}, \mathbf{p}_i) = \sum_j w_j \cdot |q_j - p_{i,j}|$$

- L_2 -norm or Euclidean Distance:

$$\delta(\mathbf{q}, \mathbf{p}_i) = \sqrt{\sum_j w_j^2 \cdot (q_j - p_{i,j})^2}$$

- L_k -norm or k -norm:

$$\delta(\mathbf{q}, \mathbf{p}_i) = \sqrt[k]{\sum_j w_j^k \cdot (q_j - p_{i,j})^k}$$

- L_∞ -norm or Maximum norm:

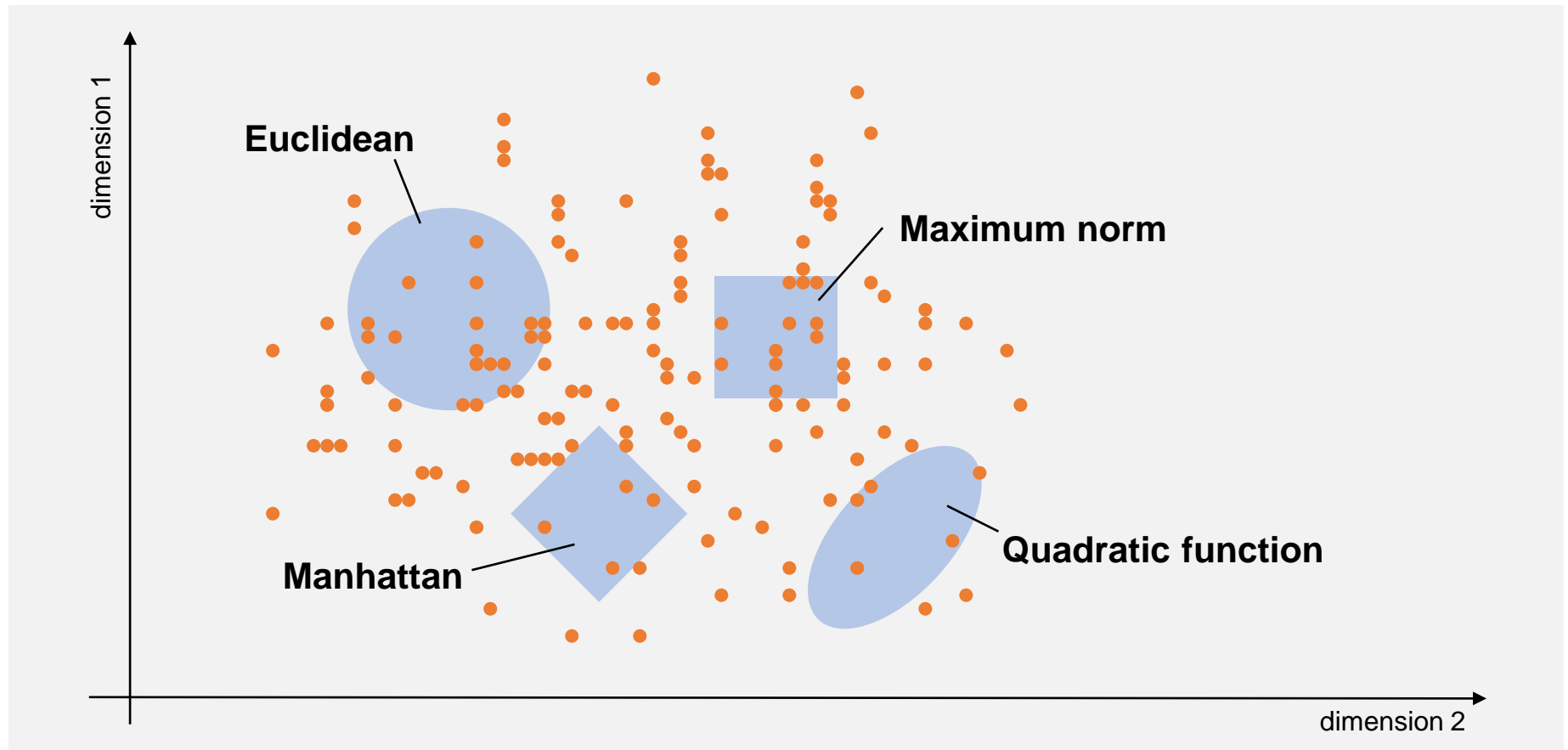
$$\delta(\mathbf{q}, \mathbf{p}_i) = \max_j (w_j \cdot |q_j - p_{i,j}|)$$

- For correlated dimensions, we can use a quadratic function with a matrix $\mathbf{A} \in \mathbb{R}^d$ that compensates correlation. In this case, weights are already factored into the correlation matrix:

- Quadratic function:

$$\delta(\mathbf{q}, \mathbf{p}_i) = (\mathbf{q} - \mathbf{p}_i)^\top \mathbf{A} (\mathbf{q} - \mathbf{p}_i)$$

- The following visualization shows all distance measures. The blue area depicts the neighborhood areas around the centers of the areas (e.g., a query vector):



– Example for weights: consider the following two dimensions

- In dimension d_1 , all values are between 0 and 1.
- In dimension d_2 , all values are between 100 and 200.

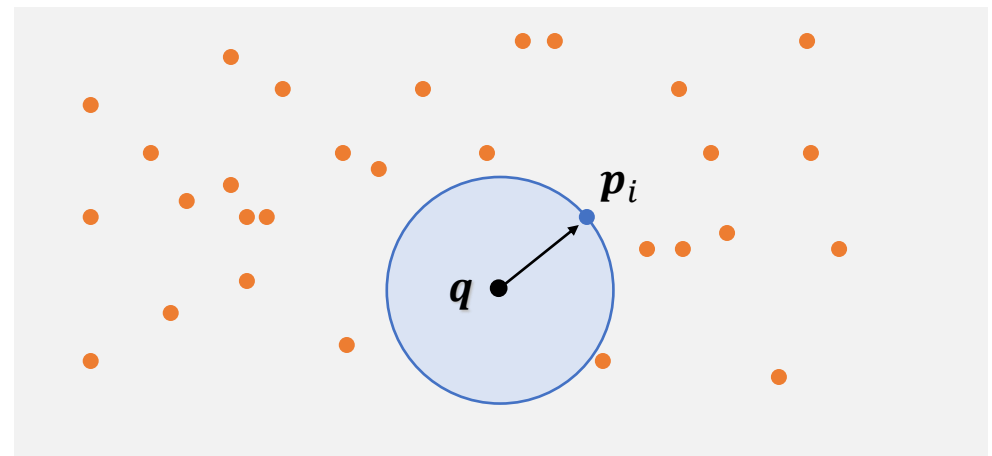
If we would apply an unweighted distance function, dimension d_2 would dominate dimension d_1 . In other words, regardless of how close the features are in dimension d_1 , only the difference in dimension d_2 really matters. Similarity is hence based (almost) entirely on dimension d_2 . With the weights, we can normalize the different ranges along dimensions. Note that all metrics are based on differences so that the absolute values do not matter if ranges are similar.

- Searching for the most similar object translates to a search for the object with the smallest distance, the so-called **nearest neighbor**. We note the reversed relationship between similarity values and distances:
 - large distances correspond to low similarity values
 - small distances correspond to high similarity values

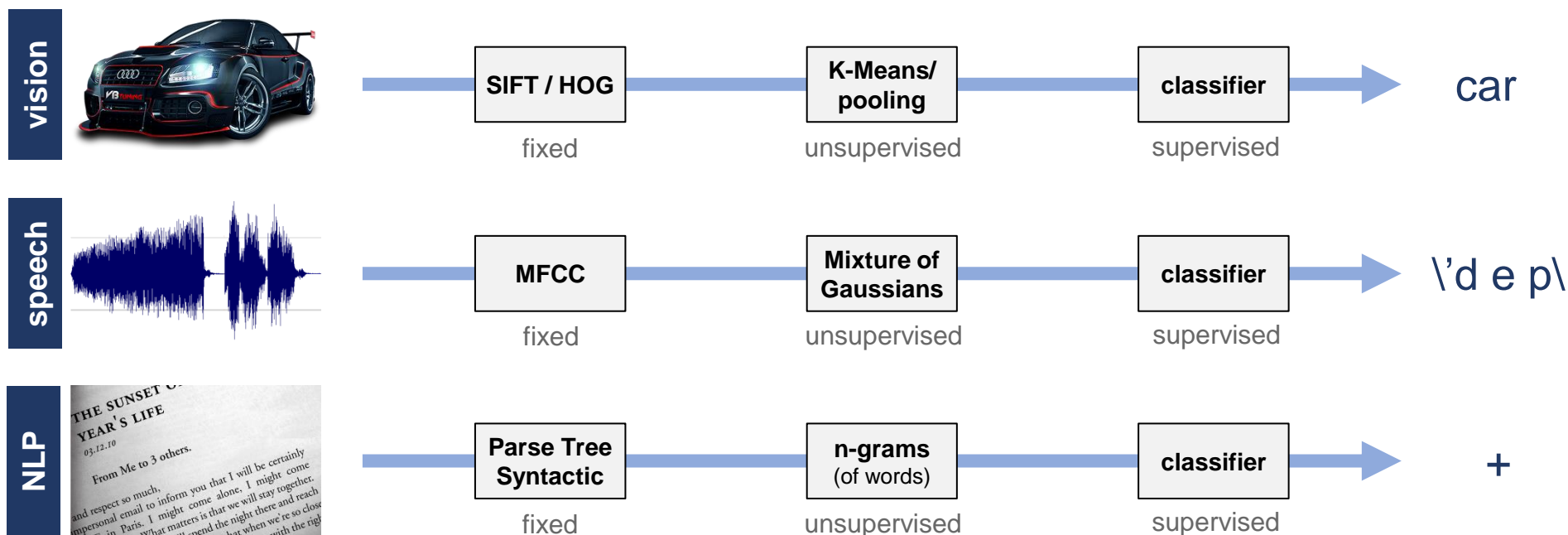
We can express similarity search as a nearest neighbor search:

Nearest Neighbor Problem:

- Given a vector q and a set \mathbb{P} of vectors p_i and a distance function $\delta(q, p_i)$
- Find $p_i \in \mathbb{P}$ such that:
$$\forall j, p_j \in \mathbb{P}: \delta(q, p_i) \leq \delta(q, p_j)$$



- Signal information is often too low level and too noisy to allow for accurate recognition of higher-level features such as objects, genres, moods, or names. As an example, there are exceedingly many ways how a chair can be depicted in an image based on raw pixel information. Learning all combinations of pixels or pixel distributions is not a reasonable approach (also consider clipped chairs due to other objects in front of them).
- Feature extraction based on machine learning abstracts lower level signal information in a series of transformations and learning steps as depicted below. The key ingredient of a learning approach is to eliminate noise, scale, and distortion through robust intermediate features and then cascade one or many learning algorithms to obtain higher and higher levels of abstractions.
- Newer approaches in deep learning even learn automatically which features to extract and how to transform features to make them more robust.

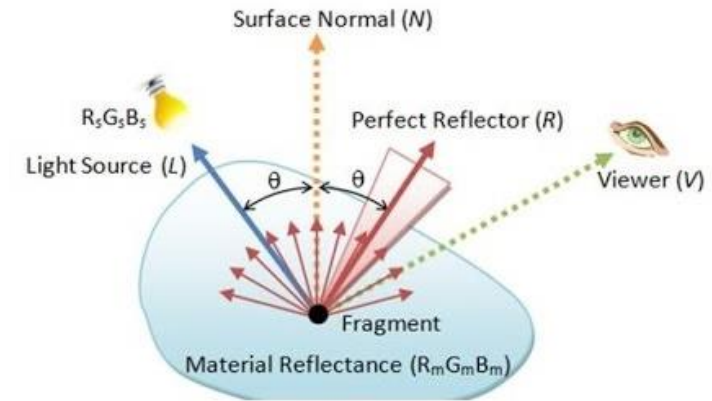


4.2 Visual Perception

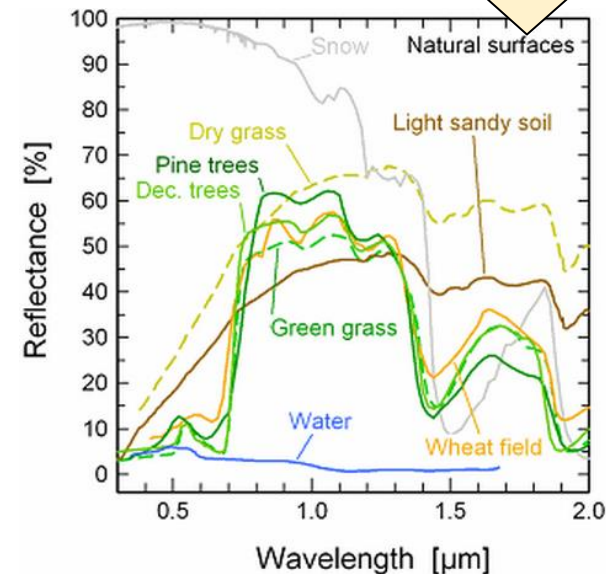
- Let's first consider how we perceive and process visual information. Perception of light is the result of illumination of an object and the amount of illumination that is reflected by the objects in front of us:
 - Illumination** $l(x, y, z)$ is the amount of lumens per square meter (=lux). Lumen is a measure of energy per second modelled along the eye's sensitivity range of light.
 - Reflectance** $r(x, y, z)$ is the amount of illumination reflected by the surface of objects. Reflectance is a function of wavelength, absorption, and direction of illumination.

Typical illuminance and reflectance values are given below:

Illuminance (lux)	Surfaces illuminated by
0.0001	Moonless, overcast night sky
0.05–0.36	Full moon on a clear night
20–50	Public areas with dark surroundings
50	Family living room lights
100	Very dark overcast day
320–500	Office lighting
400	Sunrise or sunset on a clear day.
1000	Overcast day; typical TV studio lighting
10,000–25,000	Full daylight (not direct sun)
32,000–100,000	Direct sunlight

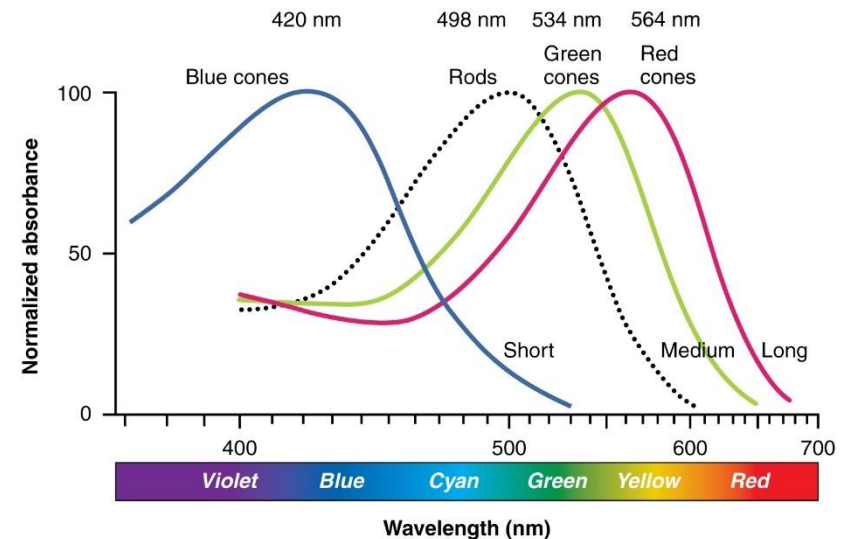
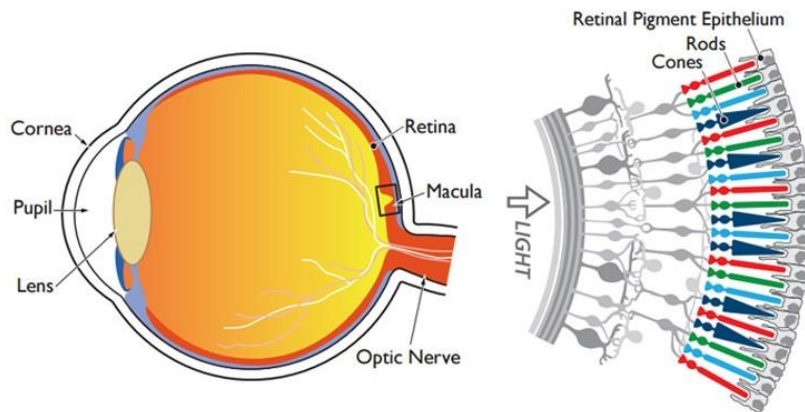


Chlorophyll has its reception peaks in the blue and red spectrum of light. Hence, we observe only the reflected green spectrum of light.

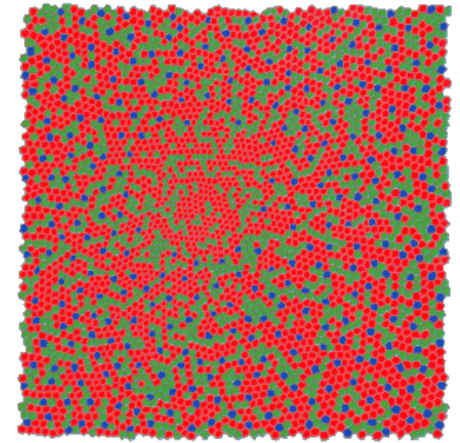


- The eye receives light and translates the wavelengths into electro-chemical impulses
 - The cornea, pupil, and lens form an adaptive optical system to focus on objects (distance) and adjust to light exposure (aperture). The lens works like an ordinary camera and projects an (upside-down) image of the world onto the retina at the back side of the eye.
 - The retina consists of three cone types and rods; they are the photoreceptors that transform incoming light energy into neural impulses. The cones enable color vision, specialize on different wavelength ranges, and are very frequent in the center of vision (macula and fovea)
 - L-cone (long wavelength) peak at 564nm corresponding to the color red
 - M-cone (medium wavelength) peak at 534nm corresponding to the color green
 - S-cone (short wavelength) peak at 420nm corresponding to color blue

The rods perform better at dimmer light and are located at the periphery of the retina. They focus on peripheral vision and night vision.

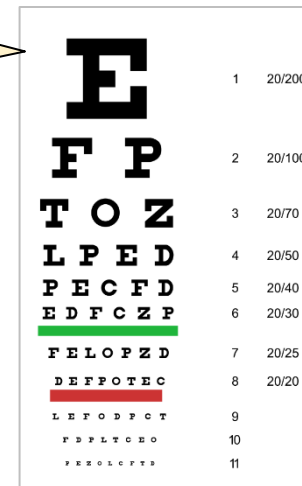


- The human eye has about 6 million cones and 120 million rods. The distribution is roughly 1% S-cones (blue), 39% M-cones (green) and 60% L-cones (red). The picture on the right shows the distribution near the center of sight (blue cones occur here up to 7%). These ratios can greatly vary and cause color blindness. Cones are focused around the fovea (see lower right side), while rods fill the periphery of sight.
- **Visual Acuity** describes the clarity of vision and how well the eye can separate small structures. With the standard Snellen chart, a 20/20 vision denotes that the eye is able, at 20 feet distance, to separate structures that are 1.75mm apart. This corresponds to roughly one arcminute (1/60 degree). A 20/40 vision denotes that a person can see things at 20 feet distance as good as a normal person at 40 feet distance. The best observed vision for humans is 20/10. Visual acuity is limited by the optical system (and defects like short-sightedness) and the number of cones and rods per mm².

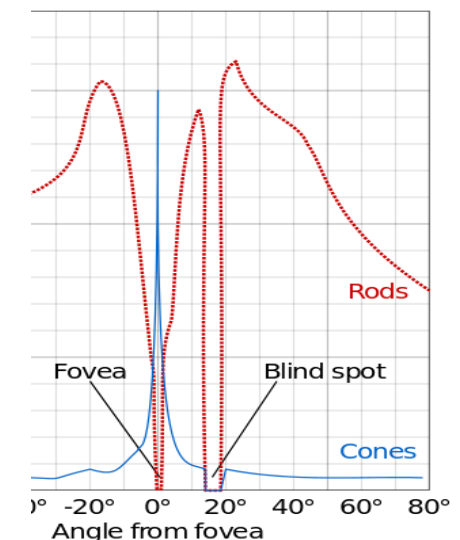


Ratio	Metric	Snellen	Arcminutes
2,0	6/3	20/10	0.5'
1,33	6/4,5	20/15	0.75'
1,0	6/6	20/20	1'
0,8	6/7,5	20/25	1.25'
0,67	6/9	20/30	1.5'
0,5	6/12	20/40	2'
0,4	6/15	20/50	2.5'
0,2	6/30	20/100	5'
0,1	6/60	20/200	10'
0,05	6/120	20/400	20'

Standard
Snellen
Chart



1.4' or less is
required to
drive a car

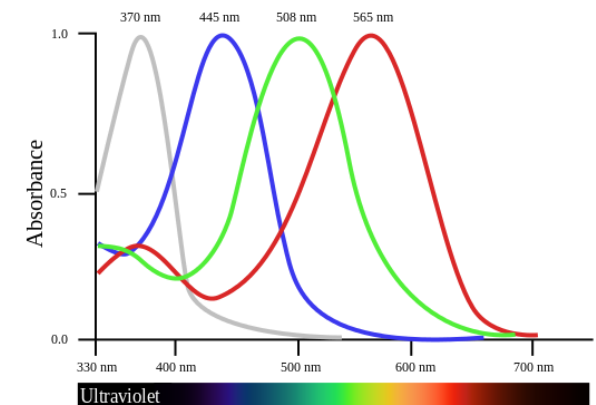


- The comparison with animals shows great differences in terms of visual sensing. A cat has a much lower visual acuity of 20/100 and less cone types (blue at 450nm and yellow at 550nm), but cats have better night vision (6-8 times) and a broader range of vision (200 degree vs 180 degree). Hence, a cat has a much blurred view compared to humans. Dogs are also dichromatic (blue/yellow) with a visual acuity of 20/75. Elephants have a 20/200 vision, rodents a 20/800 vision, bees a 20/1200 vision, and flies a 20/10800.



On the other side, eagles and bird of prey have a 20/4 vision (5 times better than the average human). In addition, some birds are tetrachromatic and see the world with four independent color channels. The goldfish and zebrafish also have four different cone types. The additional cone type is typically in the ultra-violet range with a peak at about 370nm.

- **Conclusion:** our color vision is a sensation but not physics. To understand how we perceive images, we need to follow the way the human eye (and brain) processes light.

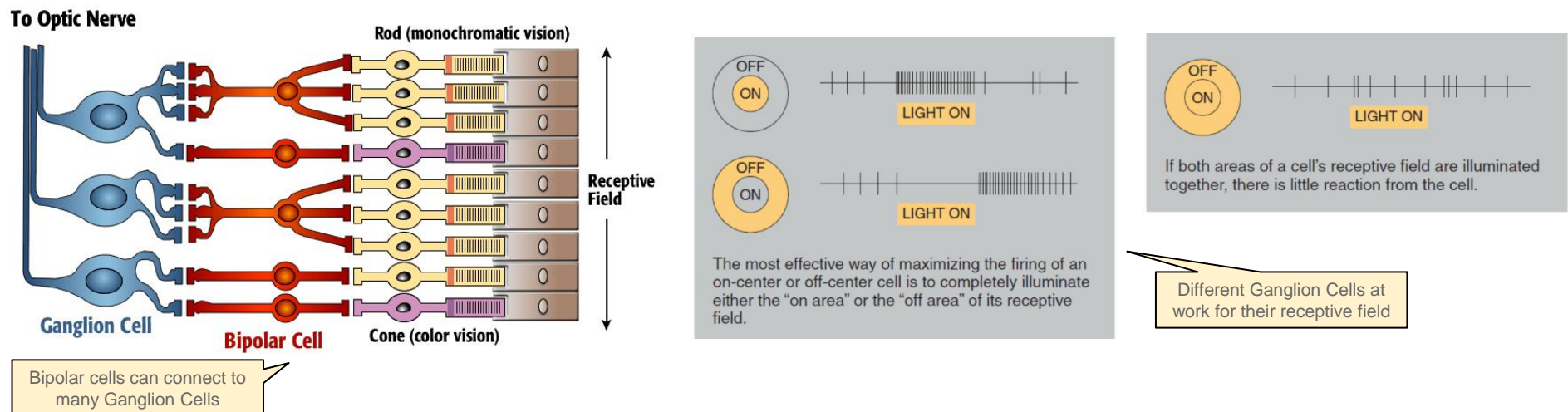


- The first processing starts within the retina (we will see similar concept in deep learning by means of convolution). The chemical process in the rods and cones release glutamate when it's dark, and stop releasing glutamate when it's light (this is unusual for a sensory system). The **Bipolar Cells** connect to several rods and cones (but never both together) and perform a simple operation:
 - On-Bipolar cells, fire when it is bright
 - Off-Bipolar cells, do not fire when it is bright

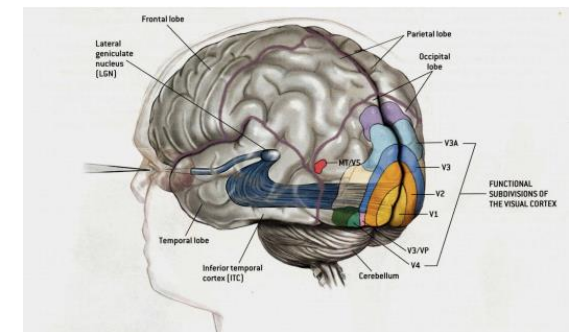
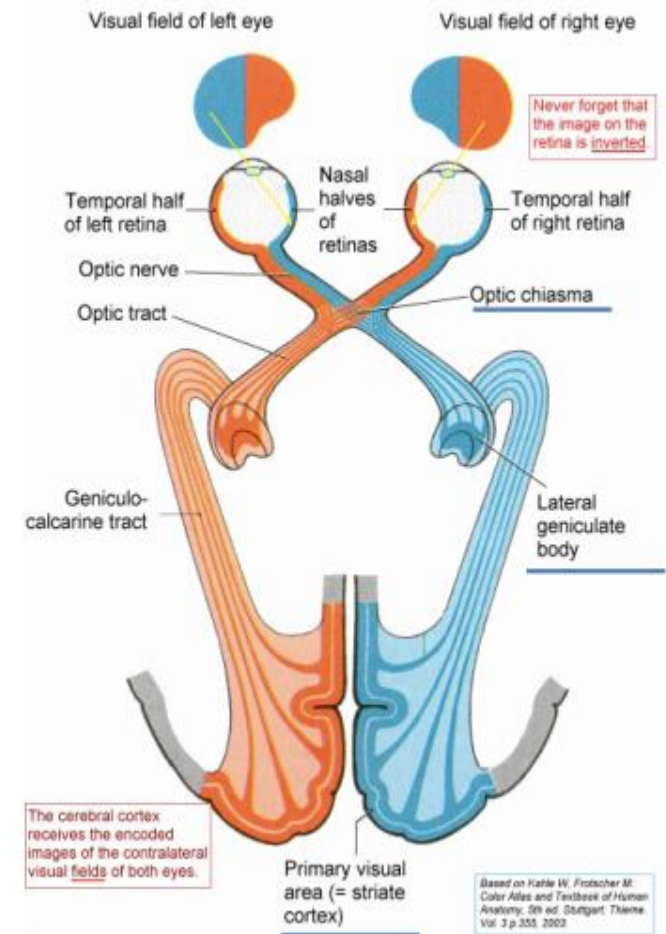
The next stage, the **Ganglion Cells** build the first receptive fields combining various bipolar cells. In a nutshell, they perform edge detection with a center and a surround area.

- On-Center ganglion fires, if center is bright and surrounding is dark
- Off-Center ganglion fires, if center is dark and surrounding is bright

Several additional cell types (horizontal cells, amacrine cells) act as inhibitors to accentuate contrast. This increased contrast can also lead to falsely under-/oversaturating dark/light boundaries. Lateral inhibition provides negative feedback to neighbor cells to further strengthen the contrast between strong and weak signals. This can lead to so-called after-images.

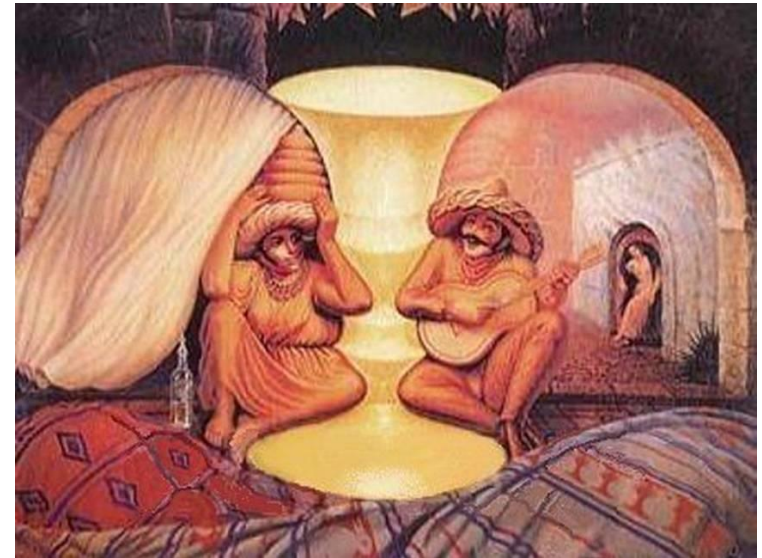
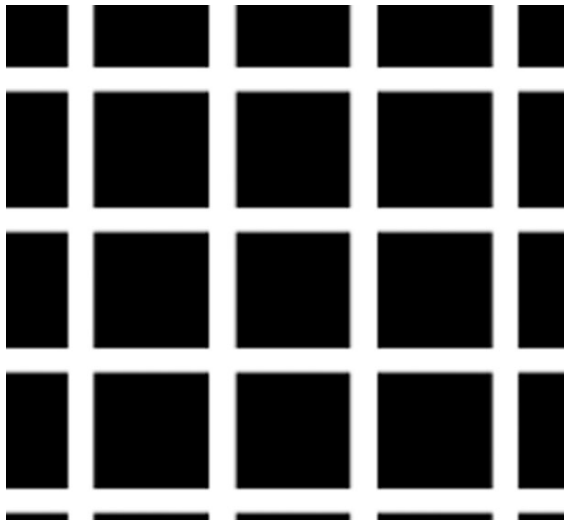
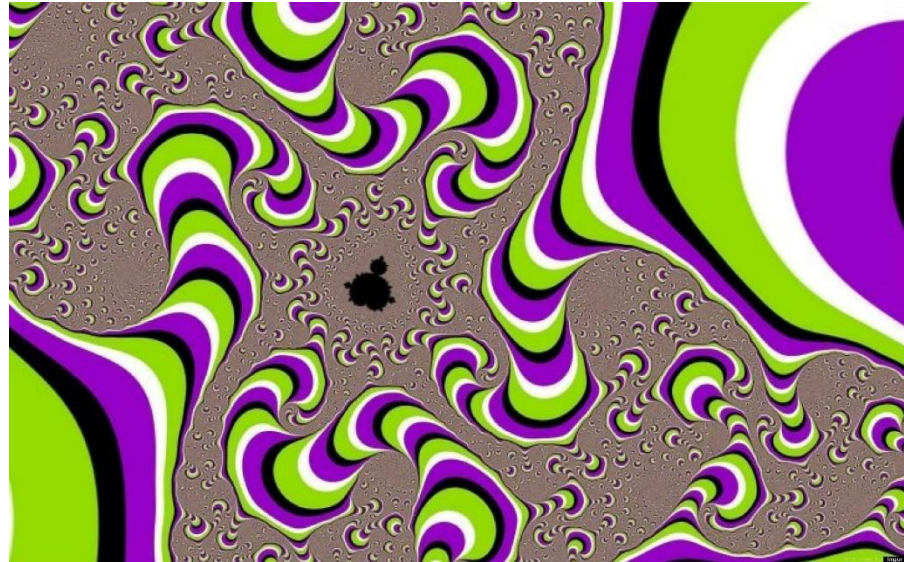
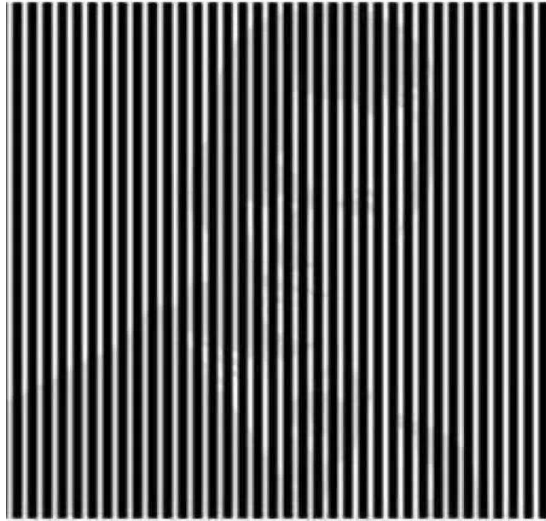


- The **Lateral Geniculate Nucleus (LGN)** performs similar receptive field functions as the ganglion cells but with massive feedback from the cortex. We first observe a split of the two visual fields (visual left is processed by the right side of the brain, visual right is processed by the left side). Secondly, the information of both eyes is combined. The first two layers focus on rods and the detection of movements and contrast. The next 4 layers process information from cones to perceive color and form (finer details).
- The **Primary Visual Cortex (V1)** performs detection of edges, orientation, some of them variant to position, others invariant to position. Neurons in the visual cortex fire when the defined patterns occur within their receptive fields. In the lower levels, the patterns are simpler; in higher levels, more complex patterns are used (e.g., to detect a face). The stream of information flows along two paths to higher levels.
 - The **Ventral Stream** (ventral=underside, belly) specializes on form recognition and object representation. It is connected with the long-term memory.
 - The **Dorsal Stream** (dorsal=topside, back) focuses on motion and object locations, and coordinates eyes, heads, and arms (e.g., reaching for an object)
- Cortical magnification denotes the fact that the majority of neurons act on the information in the center of vision (creating a much denser, magnified view of the center)



- The visual perception system is optimized for natural image recognition. Artificial illusions demonstrate very nicely how the brain processes the perceived environment in many ways:

Shake your head



4.3 Image Normalization

- In image processing, an image is usually described as a discrete function mapping a 2-dimensional coordinate to an intensity value (gray images) or a color value. We will use the function $i(x, y)$ and $\mathbf{i}(x, y)$ to denote such images:

grayscale images:	$i(x, y): \mathbb{N}^2 \rightarrow [0, 1]$
color images:	$\mathbf{i}(x, y): \mathbb{N}^2 \rightarrow [0, 1]^3 = \begin{bmatrix} r(x, y) \\ g(x, y) \\ b(x, y) \\ \alpha(x, y) \end{bmatrix}$
color channels (red)	$r(x, y): \mathbb{N}^2 \rightarrow [0, 1]$
color channels (green)	$g(x, y): \mathbb{N}^2 \rightarrow [0, 1]$
color channels (blue)	$b(x, y): \mathbb{N}^2 \rightarrow [0, 1]$
α -channel (transparency)	$\alpha(x, y): \mathbb{N}^2 \rightarrow [0, 1]$
with	$1 \leq x \leq N, 1 \leq y \leq M$

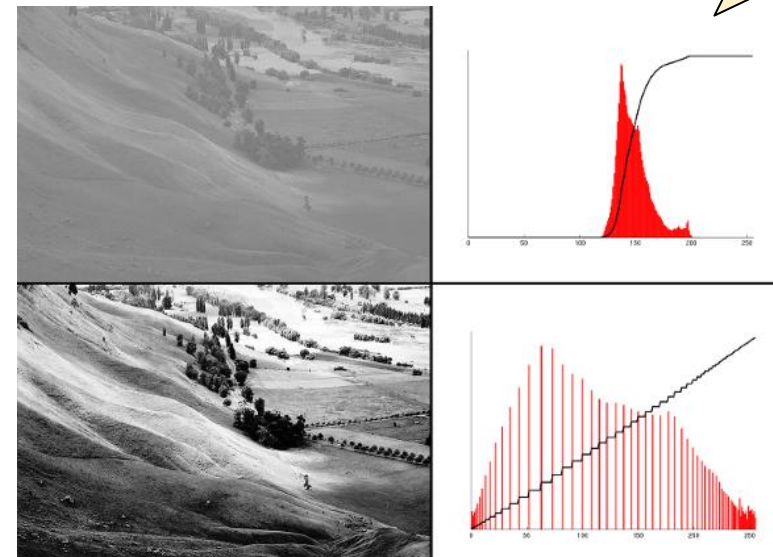
- It is custom to start with the upper left pixel ($x = 1, y = 1$) and to end with the lower right pixel ($x = N, y = M$). x denotes the row in the image (vertical axis), while y denotes the column in the image (horizontal axis).
- Quantization is often applied to avoid fixed point numbers in the image representation. Quantification is an approximation of the fixed point number as follows:

True Color (32-bit):	$\hat{f}(x, y): \mathbb{N}^2 \rightarrow [0, 255]$	approximating $f(x, y) = \frac{\hat{f}(x, y)}{255}$
Deep Color (64-bit):	$\hat{f}(x, y): \mathbb{N}^2 \rightarrow [65535]$	approximating $f(x, y) = \frac{\hat{f}(x, y)}{65535}$

f denotes one of i, r, g, b, α

- Other quantization with indexed colors exist but can be mapped to one of the above.

- Depending on the data collection, we need to perform a number of image processing steps to normalize the data sets and to achieve the best results when comparing features afterwards. Some of the processing steps ensure robustness against noise, rotation, color saturation, or brightness which are essential for the algorithms to work.
 - Rotation** – if we need rotation invariant features (texture, shape) but do not have enough information to normalize direction, we can rotate the image in defined steps of degrees, extract features, keep all features for the image, but use them as individual representation (no combination of the features). A typical approach is by 90 degrees (which makes it simple). In object recognition (faces), more intermediate angles are possible (e.g., 15 degrees)
 - Histogram normalization** – here, histogram means the distribution of brightness across the image. In poor sensing condition, the range of values can be very narrow, making it difficult to distinguish differences. Histogram equalization is the extreme case, where the range of values is forced to a uniform distribution. The picture on the right shows very nicely the increased contrast and the sharper contours of objects. With the original picture, edge detection may not lead to the expected results. Similar approaches are histogram shifts (lighter, darker), histogram spreading, or gamma correction.
 - Grayscale transformation** – The original color image is transformed to a grayscale image. Depending on the source color model, different formulae define how to calculate the gray value. Often applied before texture and shape analysis as color information is not needed.



- **Scaling** – Up- or down-sampling of the image to fit within a defined range of acceptable sizes. For instance, a neural network might expect the input to fit into the input matrix. A shape or texture feature is sensitive to different scaling and may yield different results. The usual methods are bilinear or bicubic interpolation to avoid the creation of artefacts that could negatively impact the algorithms (in combination with Gaussian filters when down-sampling). If the algorithm is complex and expensive, down sampling is often applied to reduce the efforts. In such cases, the results are computed for the down-sampled image only, and then mapped back to the original image (see k-means clustering later on for image segmentation).
- **Affine Transformation** – The generalization of translation, rotation and scaling. The original coordinates (x, y) are mapped to a new pair (x', y') as follows:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

With this matrix representation, we can simplify the concatenation of various operators to obtain a single matrix again. To improve results, bilinear or bicubic interpolation is needed to estimate pixel values in the new matrix. Note: the affine transformation above does not necessarily map to a discrete and positive coordinate systems, and some areas in the new image space may have unknown values (think about a rotation by 45 degrees mapped to minimum bounding box).

- **Noise Reduction / Sensor Adjustments** – Sensors, transcoding and digitization can add noise (think of white and black pixels across the image) that can significantly impact the feature extraction process. Common methods are mean filter or Gaussian filters as described next. Other adjustments may include color corrections, distortions, moiré patterns or compression artifacts.

- **Convolution** is a mathematical operation that combines two functions to produce a new function. It is similar to the cross-correlation but considers values “backwards” and integrates them. The discrete two-dimensional form is given as (* denotes the convolution operation)

$$(f * g)[x, y] = \sum_{n=-\infty}^{\infty} \sum_{m=-\infty}^{\infty} f[x - n][y - m] \cdot g[n][m]$$

- In image processing, g is called the Kernel and is typically a very small two-dimensional quadratic (and often symmetric) function with range $[-K, K] \times [-K, K]$ with small values $K = 1, 2, 3, 4, \dots$. Applied to an image channel $f(x, y)$ we obtain


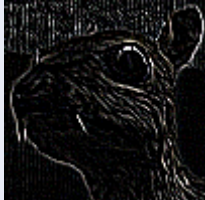

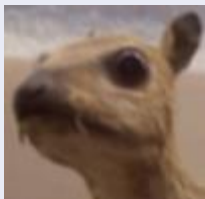
$$(f * g)[x, y] = \sum_{n=-K}^K \sum_{m=-K}^K f[x - n][y - m] \cdot g[n][m]$$

- As a visualization, assume we calculate the convolution of a 3x3 image with a 3x3 kernel for the center point of the image ($x = y = 2$). For example:

$$\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \right) [2,2] = (i \cdot 1) + (h \cdot 2) + (g \cdot 3) + (f \cdot 4) + (e \cdot 5) + (d \cdot 6) + (c \cdot 7) + (b \cdot 8) + (a \cdot 9)$$

Note that the Kernel is actually flipped horizontally and vertically and then dot-wise multiplied with each image element. If the Kernel is symmetric, we can just apply the dot-wise multiplication to compute the convolution. Further note, that the Kernel is moved with its center across the image to compute a new value for that current pixel. If the Kernel overlaps the image, we use 0-padding for pixels beyond the boundary to keep image dimensions.

- Kernel Examples: (taken from Wikipedia for illustration purposes). When defining a Kernel, it is important to normalize the output by the sum of all Kernel values, otherwise channel values may exceed the defined boundaries ([0,1] or, if quantized, [0,255]).

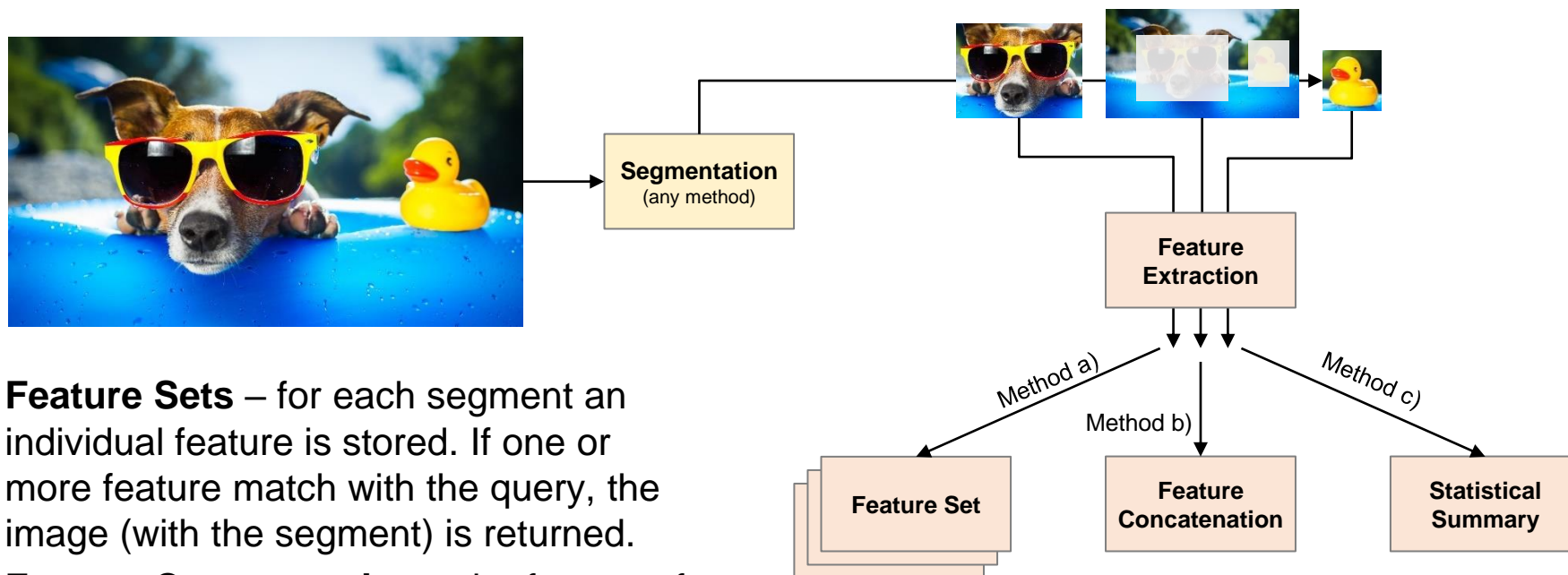
Operation	Kernel	Image Result
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge Detection	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box Blur	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	

The Kernel values do not sum up to 1. We are leaving the value range of the underlying color space and enter a new value space (derivatives)

Here, we need to divide by the sum of the Kernel values to prevent leaving the value range of the underlying color space

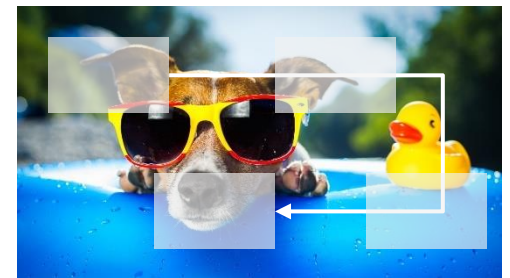
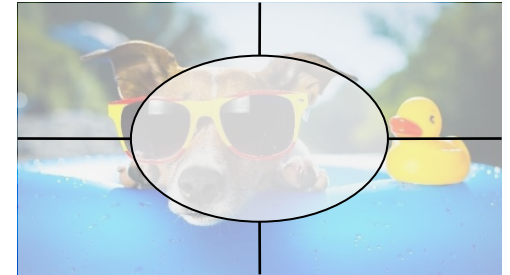
4.4 Image Segmentation

- Feature design may include the capturing of location information (much like we did with position information in text retrieval). Segmentation define areas of interest within the image for which the features are computed. To obtain overall features for the image, three different ways are possible:

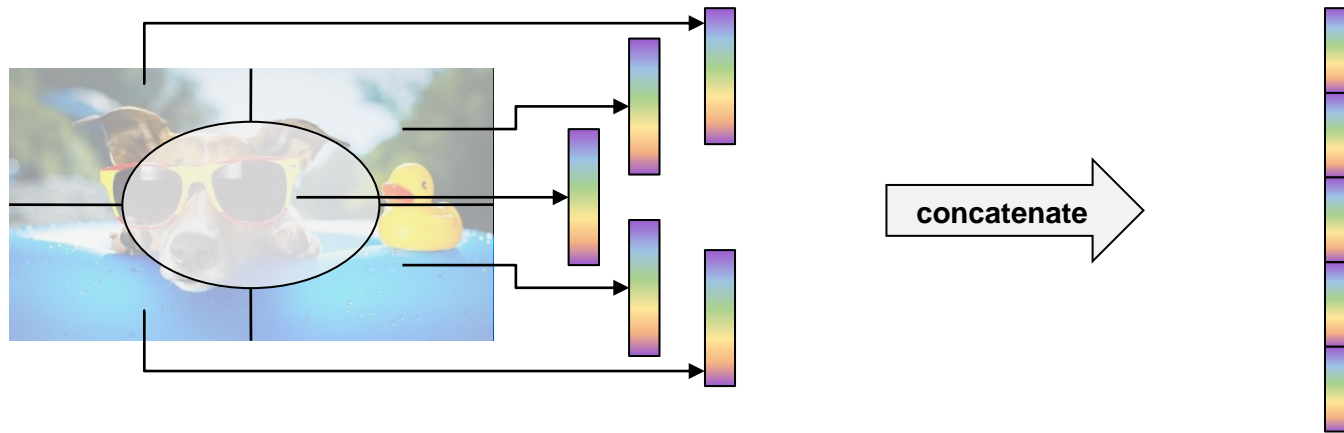


- Feature Sets** – for each segment an individual feature is stored. If one or more feature match with the query, the image (with the segment) is returned.
- Feature Concatenation** – the features for each segment are combined to form an overall feature for the image. This approach is only meaningful for pre-defined segmentations but not for object related segmentation with varying number of segments.
- Statistical Summary** – the features are summarized with statistical operators like mean, variance, co-variance, or distribution functions. The statistical parameters describe the image. If the segmentation only yields one segment (global features), all methods become identical.

- We can segment images with three approaches (actually the first one does nothing)
 - **Global** features require the entire image as input. No segmentation occurs. This approach is often the standard in absence of a clear segmentation task. We will see later that with temporal media like audio and video, global features are very rare but quite common for still images.
 - **Static Segmentation** uses a pre-defined scheme to extract areas of interest from the image. There are two reasons for such a segmentation
 - Add coarse location information to the features. Typically, an image consists of a central area (the object) and four corner areas (as shown on the right). But any type of regular and potentially overlapping division is possible. Often, this method is combined with the concatenation of features to encode left/right, up/down, or center within the feature.
 - Process parts of the query image to detect similar features. We use a sliding window that moves from upper left to lower right in defined steps. For each position, features are extracted and used to find matches. For example, when detection faces the sliding window technique allows to find many faces together with their location from a given input picture (see next chapter).
 - **Object Segmentation** extracts areas with embedded objects in the picture (so-called blobs). These blobs are either analyzed individually or as a part of the image. Often, feature sets are used to enable individual retrieval of the blobs. We will study such an approach in the next chapter (k-means clustering).



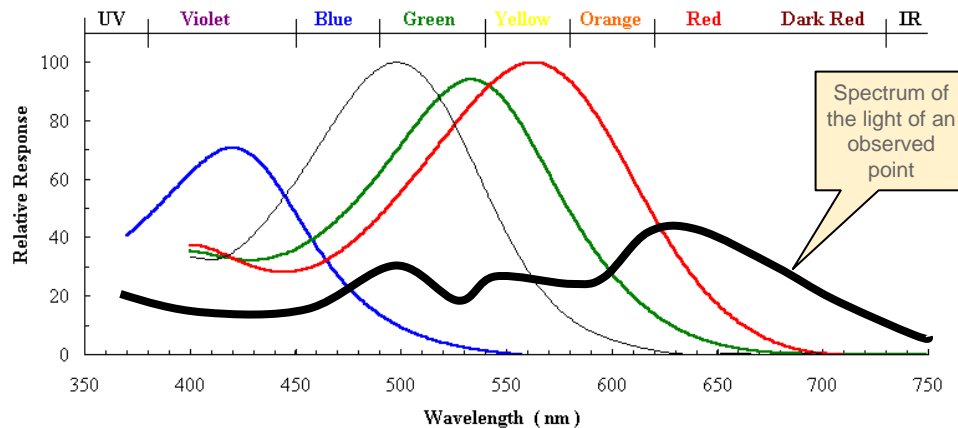
- Example: 9-dimensional color feature with 5 static segments
 - Segmentation creates 5 areas for each of which a 9-dimensional feature is extracted



- The feature for the image has 45-dimensions and encode localized color information. To be similar with the above picture, the colors not only have to occur in a similar way but they also have to be in the same area. On the other side, we loose some invariances, like rotation. An upside-down version of the picture does not match with itself. On the other side, a blue lake does not match with the blue sky, a white background (snow) does not match with the white dress (center), and an object on the left does not match with the same object on the right.
- We will see, that a single feature is often not sufficient to find similar pictures. Rather, we need to construct several (very similar) features to encode the different choices for variance and invariance. Segmentation, obviously, can both eliminate location information (for instance feature sets), enforce location (feature concatenation), or is liberal about the position (statistical summary and feature set).

4.5 Color Information

- We split the third step, feature extraction, into color, texture and shape information. We start with color in this subsection.
- Color perception is an approximation of the eye to describe the distribution of energy along the wavelength of electromagnetic signals. “Approximation” because the distribution cannot be described accurately with only 3 values, hence most information is lost. It is possible to construct two different spectra which are perceived exactly the same.

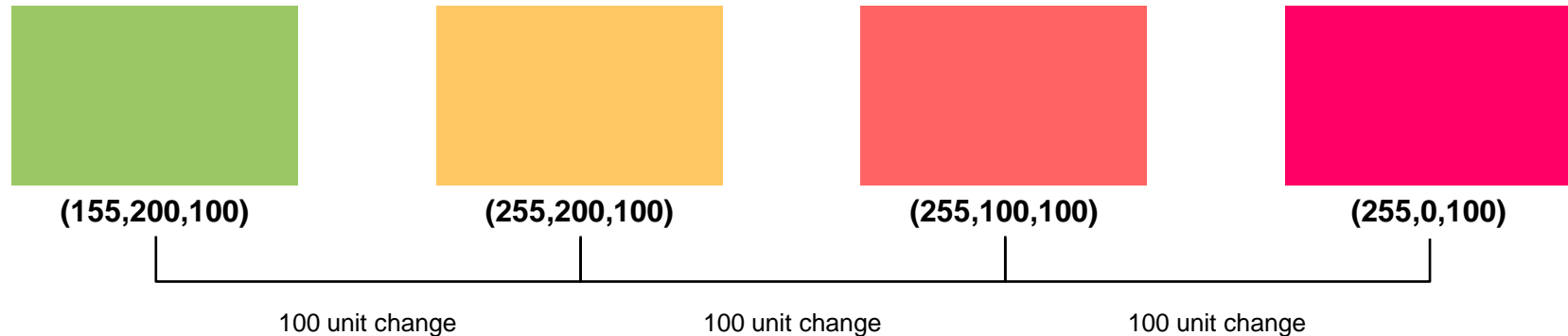


Given the emitted or reflected spectrum of light of an observed point $f(\lambda)$, we perceive 3 (4) values for each cone type (and rod). To compute the intensity, we apply the sensitivity filter of the cones (e.g., $c_{red}(\lambda)$) to the observed spectrum (multiplication) and integrate the result over all wavelengths. For instance, for red this is:

$$red = \int_0^{\infty} f(\lambda) \cdot c_{red}(\lambda) d\lambda$$

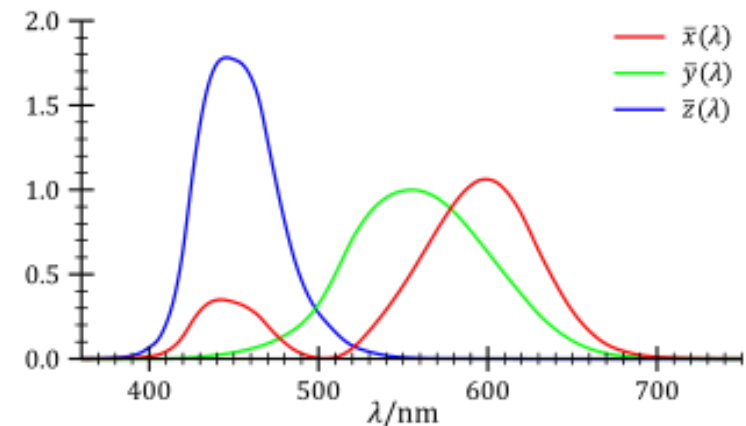
- On the other side, this approximation allows us to artificially re-create the perception with using only 3 additive components emitting wavelengths that match the sensitivity of the red, green, and blue cones. These 3 components form the basis of the RGB family which is optimized for human perception but may not work for the eyes of animals (different sensitivity ranges; for birds with tetrachromatic perception, the UV range is missing).

- Before we can extract features, we need to find a good representation for color that matches human perception. Consider the four colors below in the sRGB space. Between two neighboring boxes, the color distance is 100 units (only one channel changes). Even though the distance is the same, we perceive the color changes differently. The change from green to yellow (1st and 2nd) is significant, while the change from red to pink (3rd to 4th) is smaller. The reason is the non-linear interpretation of sRGB space as we process the light emission from the monitor (or from the reflection of the paper).



- There are five major color systems (we only look at the first three models subsequently)
 - CIE** – created by the International Commission on Illumination (CIE) to define a relation between the physical signal and the perception of a (standard) human observer
 - RGB** – the dominant system since the definition of sRGB by HP and Microsoft in 1996
 - HSL/HSV** – which translates the cartesian RGB coordinates to cylindrical coordinates for hue and saturation, and uses luminance/brightness as third component
 - YUV** – used in NTSC and PAL signals and basis of many image and compression algorithms such as JPEG and MPEG (using YCbCr) *[not discussed subsequently]*
 - CMYK** – used in printing to subtract color from an initially white canvas. The ink absorbs light and a combination of different inks produces the desired color *[not discussed subsequently]*

- The CIE defined a series of color spaces to better describe perceived colors of human vision. The mathematical relationships are essential for advanced color management.
 - The **CIE XYZ** space was defined in 1931 as an attempt to describe human perceived colors. In their experiments, they noted that observers perceive green as brighter than red and blue colors with the same intensity (physical power). In addition, in low-brightness situations (e.g., at night) the rods dominate with a monochromatic view but at much finer resolution of brightness changes.
 - The definition of X , Y and Z does not follow the typical approach of additive or subtractive primary colors. Instead, Y describes the luminance while X and Z describe chromaticity regardless of brightness. Y follows the sensitivity for the M-cones (green), Z the one of the S-cones (blue), and X is a mix of cone responses.
 - To compute X , Y , and Z from spectral data, a standard (colorimetric) observer was defined based on extensive experiments. This represents an average human's chromatic response within a 2 degree arc inside the fovea (central vision; cones mostly reside inside this area). The color matching functions $\bar{x}(\lambda)$, $\bar{y}(\lambda)$ and $\bar{z}(\lambda)$ describe the spectral weighting for the observed spectral radiance or reflection $f(\lambda)$. We obtain the values for X , Y , and Z as follows (note that the spectrum is reduced to the range 380nm to 780nm):

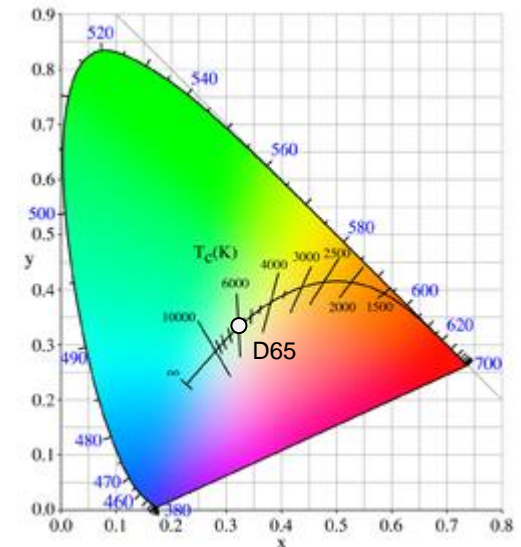


$$X = \int_{380}^{780} f(\lambda) \cdot \bar{x}(\lambda) d\lambda$$

$$Y = \int_{380}^{780} f(\lambda) \cdot \bar{y}(\lambda) d\lambda$$

$$Z = \int_{380}^{780} f(\lambda) \cdot \bar{z}(\lambda) d\lambda$$

- The three cone types of human vision require 3 components to describe the full color gamut. The concept of color can be divided into different aspects:
 - Brightness – visual perception of the radiating or reflected light and dependent on the luminance of the observed object. It is, however, not proportional to the luminance itself, instead it is an interpretation subjective to the observer.
 - Chromaticity – objective specification of the color in absence of luminance. It consists of two independent components, hue and saturation. Chromaticity diagrams depict the visible or reproducible range of colors. The standard chart is depicted on the right side.
 - Hue – describes the degree a color matches the perception of red, green, blue, and yellow. The hue values are on the boundary of the chromaticity diagram and is usually measured as a degree from the neutral white point (e.g., D65). Red corresponds to 0, yellow to 60, green to 120, and blue to 240.
 - Saturation / Chroma / Colorfulness – measure how much the light is distributed across the visual spectrum. Pure or saturated colors focus around a single wavelength at high intensity. To desaturate a color in a subtractive system (watercolor), one can add white, black, gray, or the hue's complement. In the chromaticity diagram, saturation is the relative distance to the white point. Relative means in terms of the maximum distance in that direction. Note that green is much farther away from white than red and blue.
- The CIE then defined a series of color models to better capture the above components of color perception. We consider in the following the CIE xyY, Lab, and LCH model.



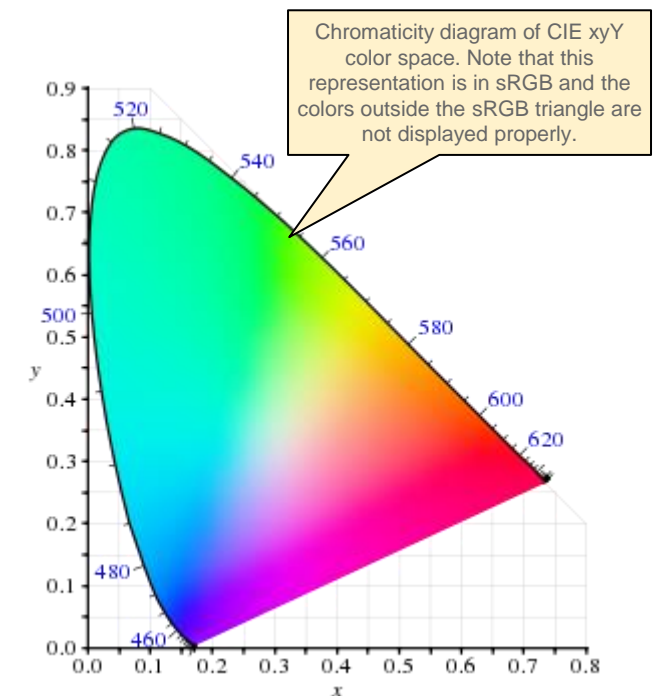
- The CIE xyY space, defined in 1931, was the first attempt to isolate chromaticity from luminance. The Y value of CIE XYZ was created in such a way that it represents perceived luminance of the standard observer. The x , y and z components are derived through a normalization

$$x = \frac{X}{X + Y + Z} \qquad y = \frac{Y}{X + Y + Z} \qquad z = \frac{Z}{X + Y + Z} = 1 - x - y$$

The derived color space consists of x , y , and Y . The x , y values define the chromaticity diagram as shown in the lower right part of the page (color in absence of luminance). CIE xyY is widely used to specify color. It encompasses all visible colors of the standard observer. Note that the picture of the chromaticity diagram here is depicted in the sRGB space and hence does not show the full gamut of the space. Given the x , y and Y values, the back transformation is as follows:

$$X = \frac{Y}{y}x \qquad Z = \frac{Y}{y}(1 - x - y)$$

The outer curve of the chromaticity diagram, the so called spectral locus, show wavelengths in nanometer. The CIE xyY space describes color as perceived by the standard observer. It is not a description of the color of an object as the perceived color of the object depends on the lightning and can change depending on the color temperature of the light source. In dim lightning, the human eye loses the chromaticity aspect and is reduced to a monochromatic perception.



- CIE xyY spans the entire color gamut that is visible for a human eye, but it is not perceptually uniform: the perceived difference between two colors with a given distance apart greatly depends on the location in the color space. The **CIE L*a*b*** color space is a mathematical approach to define a perceptually uniform color space. It exceeds the gamut of other color spaces and is device independent. Hence, it is frequently used to map color from one space to another.
 - The L component denotes lightness. It depends on the luminance Y but adjusted to perception to create a uniform scale (1 unit difference is perceived as the same lightness change). It typically ranges between 0 and 100, with $L = 0$ representing black, and $L = 100$ being white.
 - The a^* component represents the red/green opponents. Negative values correspond to green, while positive values correspond to red. The values often range from -128 to 127. $a^* = 0$ denotes a neutral gray.
 - The b^* component represents the blue/yellow opponents. Negative values correspond to blue, while positive values correspond to yellow. The values often range from -128 to 127. $b^* = 0$ denotes a neutral gray.

The transformation from X , Y , Z components under illuminant D65 and $0 \leq Y \leq 255$ is:

$$L^* = 116 \cdot f\left(\frac{Y}{Y_n}\right) - 16$$

$$a^* = 500 \cdot \left(f\left(\frac{X}{X_n}\right) - f\left(\frac{Y}{Y_n}\right) \right)$$

$$b^* = 200 \cdot \left(f\left(\frac{X}{X_n}\right) - f\left(\frac{Z}{Z_n}\right) \right)$$

$$f(t) = \begin{cases} \sqrt[3]{t} & \text{if } t > \left(\frac{6}{29}\right)^3 \\ \frac{841 \cdot t}{108} + \frac{4}{29} & \text{otherwise} \end{cases}$$

$$X_n = 242.364495$$

$$Z_n = 277.67358$$

$$Y_n = 255.0$$

- The **CIE LCH** differs from CIE $L^*a^*b^*$ by the use of cylindrical coordinates. $L = L^*$ remains, but a^* and b^* are replaced by the chroma C (saturation, colorfulness) and hue H . Based on the definition of the a^* - and b^* -axis, the center is at the defined white point (e.g., D65). The hue H is then the angle from the a^* -axis (counterclockwise). The chroma C is the distance from the center.

$$L = L^*$$

$$C = \sqrt{(a^*)^2 + (b^*)^2}$$

$$H = \arctan(a^*, b^*)$$

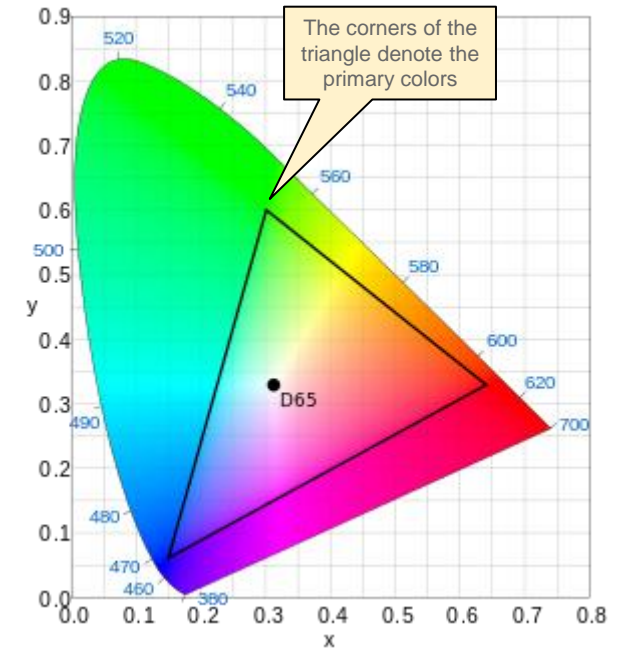
$\arctan(a^*, b^*)$ is the arc tangent of b^*/a^* taking the quadrant of (a^*, b^*) into account

- This is not the same as the better known HSL/HSV color models (also use cylindrical coordinates). These models are a polar coordinate transformation of the RGB color space, while CIE LCH is a polar coordinate transformation of CIE $L^*a^*b^*$.
- CIE LCH is still perceptually uniform. However, H is a discontinuous function as the angle abruptly changes from 2π to 0. This can cause some issues if the angles are not correctly “subtracted” from each other.
- The CIE has defined further models like the CIE $L^*u^*v^*$, CIE RGB, and the CIE UVW which we omit here.

- The **RGB** color space is the standard model in computing since HP and Microsoft cooperatively defined sRGB as an additive color model for monitors, printers and the Internet. It has been standardized as IEC 61966-2-1:1999 and is the “default” color model (if the model is not defined).
 - sRGB uses the ITU-R BT.709 (or Rec. 709) primaries to define the color gamut (space of possible colors). The advantage, and mostly the reason for its success, was the direct transfer to a typical CRT monitor at that time. The primaries are:

Chromaticity	Red	Green	Blue	White Point (D65)
x	0.6400	0.3000	0.1500	0.3127
y	0.3300	0.6000	0.0600	0.3290
Y	0.2126	0.7152	0.0722	1.0000

- For non-negative values, sRGB colors are bound to the triangle depicted in the right-hand figure. Note that the color gamut is not covering all chromaticities, especially a large fraction of the green/blue range is missing.
- The sRGB scales are non-linear (approximately a gamma of 2.2). To convert from linear RGB to sRGB, the specification provides functions to map channel values. Let c_{sRGB} denote a channel value (red, green, blue) in the sRGB space, and c_{linear} denote a value in linear RGB. Both with ranges between 0 and 1 (for quantized value, divide/multiply by $2^{\text{bits}} - 1$)

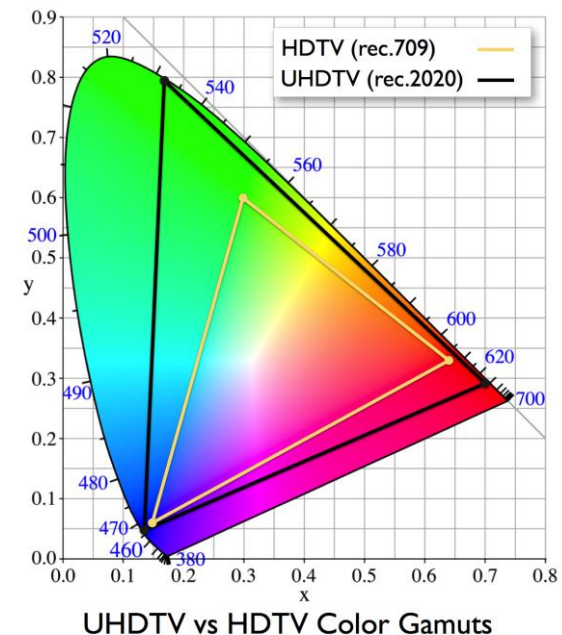


$$c_{sRGB} = \begin{cases} 12.92 \cdot c_{linear} & \text{if } c_{linear} \leq 0.0031308 \\ 1.055 \cdot c_{linear}^{\frac{1}{2.4}} - 0.05 & \text{otherwise} \end{cases} \quad c_{linear} = \begin{cases} \frac{c_{sRGB}}{12.92} & \text{if } c_{sRGB} \leq 0.04045 \\ \left(\frac{c_{sRGB} + 0.055}{1.055} \right)^{2.4} & \text{otherwise} \end{cases}$$

- The conversion from CIE XYZ to linear RGB is as follows:

$$\begin{bmatrix} r_{linear} \\ g_{linear} \\ b_{linear} \end{bmatrix} = \begin{bmatrix} 3.240479 & -1.537150 & -0.498535 \\ -0.969256 & 1.875992 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \quad \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \begin{bmatrix} r_{linear} \\ g_{linear} \\ b_{linear} \end{bmatrix}$$

- Note that the transformation above is a mapping between linear RGB and XYZ. To obtain sRGB values, a further transformation is needed (see previous page).
- Also note that the RGB space is not covering the entire XYZ space and the visible colors of human perception. If the mapping leads to values outside of $[0,1]$, the value is mapped to the closest limit (0 for negative values, and 1 for values ≥ 1).
- RGB values are often quantized to integer ranges. The mapping is simply a multiplication and division by $2^{\text{bits}} - 1$. For true color (32-bit), the multiplier is 255, for deep color (64-bit), the multiplier is 65536. In some cases, quantization is based on 2^{bits} reference colors (color palette). A color is then represented by its nearest neighbor in the palette.
- Next to the sRGB and linear RGB model, various alternatives were defined. In essence, it is simple to construct an RGB space by defining the primaries and the white point. Alternative RGB model extend the original, rather constrained sRGB to a wider range of color gamut. For instance, Rec. 2020 for ultra-high-definition television (UHDTV). It has a much broader color gamut than HDTV which is based on Rec. 709. Some RGB models even exceed the chromaticity chart to cover more of the green/blue area.



- Artists often start with a relatively bright color and then add a) white to “tint” the color, or b) black to “shade” the color, or c) white and black (gray) to tone the color. To enable such techniques in computer graphics, **HSL** and **HSV** color models are alternative representations of the RGB space designed to simplify color making. Both use hue (H) and chroma (S) to define chromaticity. The HSL uses lightness (L) and places fully saturated colors at $L = 1/2$. It allows both tinting ($L \rightarrow 1$) and shading ($L \rightarrow 0$) without change of saturation. HSV uses value (V) and places fully saturated colors at $V = 1$. It allows shading ($V \rightarrow 0$) without changing saturation, but tinting adjusts saturation.

$$H' = \begin{cases} 0 & \text{if } C = 0 \\ \frac{G - B}{C} \bmod 6 & \text{if } M = R \\ \frac{B - R}{C} + 2 & \text{if } M = G \\ \frac{R - G}{C} + 4 & \text{if } M = B \end{cases}$$

$$M = \max(R, G, B)$$

$$m = \min(R, G, B)$$

$$C = M - m$$

$$H = 60^\circ \cdot H'$$

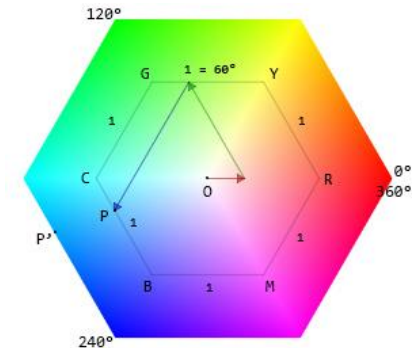
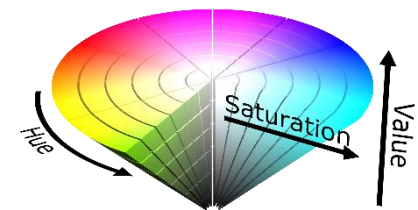
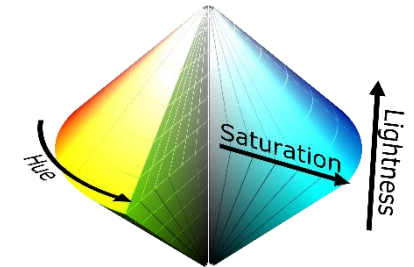
$$V = M$$

$$S_{HSV} = \begin{cases} 0 & \text{if } V = 0 \\ \frac{C}{V} & \text{otherwise} \end{cases}$$

$$H = 60^\circ \cdot H'$$

$$L = \frac{1}{2}(M + m)$$

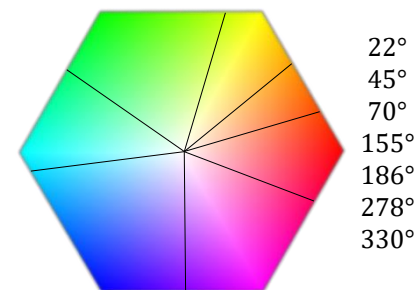
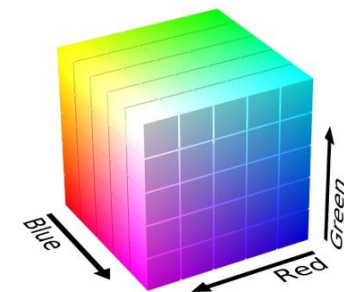
$$S_{HSL} = \begin{cases} 0 & \text{if } L = 1 \\ \frac{C}{1 - |2L - 1|} & \text{otherwise} \end{cases}$$



- **Color Histogram:** histograms are a simple way to describe the distribution of colors using a set of reference colors. The fixed reference colors are the “vocabulary” of the collection. The color of each pixel is mapped to the nearest reference color, then we count how often the reference colors occur in the image. To make the feature scale invariant, the counts are normalized by the total number of pixels. The result can also be interpreted as the probability that a reference color occurs.

- Selection of reference colors

- The most simple way is to quantize the R, G, B values in the linear RGB space as on the right hand side. With 2 bits, for example, we obtain 4 uniform ranges along each channel, and a total of 64 reference colors c_i with $1 \leq i \leq 64$. We can use any number of uniform ranges (e.g., 5) to obtain the desired number of colors.
- To improve perceptual matching of color, it is better to use a non-uniform distribution. For instance, in the HSV color space, we can divide the color hexagon into areas of perceived similar colors like on the right side. The V-dimension may have more bins to account for the increased brightness sensitivities. With 7 chromaticity values and 9 bins along the V-dimension, we obtain 63 reference colors c_i .
- If the color space itself is uniform, like in $L^*a^*b^*$, then we can use uniform ranges. The L^* -axis should have more ranges than the a^* - and b^* -axis to account for brightness sensitivity.
- We can measure the distance $d_{i,j}$ between reference color c_i and c_j to denote similarities between colors. In cartesian coordinates, this is the Euclidean distance between the centers of the areas representing the colors. In cylindrical coordinates, like the HSV example above, we obtain angle differences as $\min(|\alpha - \beta|, 2\pi - |\alpha - \beta|)$ and apply a Manhattan distance. In all cases, value ranges have to be normalized before distance calculations (e.g., to range $[0,1]$)



– Comparison of histogram (distance measure)

- Let h_i and g_i denote the normalized histograms of two images ordered by the N reference colors c_i with $0 \leq h_i, g_i \leq 1$. Note that even though we use a 3-dimensional color space for quantization, the histograms are one-dimensional (through enumeration of reference colors). We also have the distances $d_{i,j} = d_{j,i}$ between two reference colors c_i and c_j .
- A first naïve approach is to compute a Manhattan (or Euclidean) distance between histograms

$$\delta_{\text{Manhattan}}(\mathbf{h}, \mathbf{g}) = \sum_{i=1}^N |h_i - g_i| \qquad \delta_{\text{Euclidean}}(\mathbf{h}, \mathbf{g}) = \sqrt{\sum_{i=1}^N (h_i - g_i)^2}$$

This distance formulae work quite well, however, they do not take similarity between reference colors into account. A small shift in lightning or color representation can yield large distances.

- To account for cross-correlation between reference colors, we need to use a quadratic distance measure and use a matrix \mathbf{A} which is based on the distance between reference colors:

$$\delta_{\text{quadratic}}(\mathbf{h}, \mathbf{g}) = (\mathbf{h} - \mathbf{g})^T \mathbf{A} (\mathbf{h} - \mathbf{g}) \qquad \mathbf{A}: a_{i,j} = 1 - \frac{d_{i,j}}{\max_{k,l} d_{k,l}}$$

Distance normalized by maximum distance for all pairs of reference colors

- If the user provides a sketch as the query, or the user selects a number of colors that should be present in the picture, histogram intersections (equals to a partial match query) are better suited. Let $g_i \neq 0$ denote the user selected colors and $g_i = 0$ the colors without user input.

$$\delta_{\text{intersection}}(\mathbf{h}, \mathbf{g}) = \frac{\sum_{i=1}^N \min(h_i, g_i)}{\min(|\mathbf{h}|, |\mathbf{g}|)}$$

- Variants:
 - A simpler variant is the use of luminance or brightness histograms. The chromaticity aspects are not taken into account. As a first step, brightness or luminance is calculated, for instance, with L^* from CIE $L^*a^*b^*$. The luminance value is quantized using N uniform ranges. The rest is identical to the approaches above (including quadratic distances to account for similarities between brightness/luminance values). The resulting features describe brightness of the image and is often used for shot detection in videos (different lightning denotes shot boundary)
 - Equally, we can only quantize the chromaticity aspects and disregard brightness/luminance. Candidate color spaces are CIE L^*a^*b , CIE LCH, HSL, or HSV. The resulting features describes color distribution and is invariant to lightning (as long as the lightning does not significantly impact the perception of chromaticity).
- Discussion:
 - Histograms are very simple and yield already good results. They are robust against translation, rotation, noise, and scale; in some cases, also against lightning differences.
 - The lack of spatial relation between colors may lead to unexpected results. A blue lake (bottom of the picture) will match with a blue sky (top of the picture) and a blue car (middle of the picture). It is simple to construct two images with the same histogram but different content.
 - The histogram intersection method is useful to guide a retrieval system to the desired color of (main) objects. The user can pick a color and the search is extended with a histogram sub-query using the intersection method.
 - Color histograms tend to have a very high-dimensionality. 64 dimensions is often a minimum for good retrieval, but more than 1000 dimensions can result. Search in such spaces is costly and inefficient. Dimensionality reduction may help to deal with both correlation of reference colors and the reduction of dimensions (see principal component analysis, PCA).

- **Color Moments:** statistical moments are another way to describe the distribution of colors in the selected color space. We can select any of the color spaces discussed before, but again, to calculate distances and similarities, the perceptual uniform spaces are better suited. We often use $L^*a^*b^*$ as the basis color model (over LCH to avoid the more complicated angular differences)
 - Single channel moments compute statistical parameters for one channel only (L^* , a^* , b^*). Let c denote a color channel, N denote the number of rows in the image, and M the number of columns, then the first four moments are given as:

$$\mu_c = \frac{1}{N \cdot M} \sum_{x,y} c(x,y)$$

$$v_c = \frac{1}{N \cdot M} \sum_{x,y} (c(x,y) - \mu_c)^2$$

$$s_c = \frac{1}{N \cdot M} \sum_{x,y} \left(\frac{c(x,y) - \mu_c}{\sqrt{v_c}} \right)^3$$

$$k_c = \frac{1}{N \cdot M} \sum_{x,y} \left(\frac{c(x,y) - \mu_c}{\sqrt{v_c}} \right)^4$$

Mean μ_c and variance v_c describe the peak position and width of the peak in the distribution. The skewness s_c describes whether peak is wider to the left or to the right. And Kurtosis k_c denotes the presence of outliers (far away from mean). With three channels, we obtain 12 feature values in this way.

- We can add additional covariance values between pairs of channels. Let c_1 be a first channel, and c_2 be a second channel. With three channels, we obtain 3 additional covariance value from the possible pairs of channels:

$$cov_{c_1, c_2} = \frac{1}{N \cdot M} \sum_{x,y} (c_1(x,y) - \mu_{c_1}) \cdot (c_2(x,y) - \mu_{c_2})$$

- When calculating the moments, it is possible to transform the formulas such that only one pass is necessary to compute all the values (c denotes a color channel):

$$a_{c,n} = \frac{1}{N \cdot M} \sum_{x,y} c(x,y)^n \quad 1 \leq n \leq 4$$

$$b_{i,j} = \frac{1}{N \cdot M} \sum_{x,y} (c_i(x,y) \cdot c_j(x,y)) \quad 1 \leq i,j \leq 3$$

$$\mu_c = a_{c,1}$$

$$v_c = a_{c,2} - a_{c,1}^2$$

$$s_c = \frac{a_{c,3} - 3a_{c,2} \cdot a_{c,1} + 2a_{c,1}^3}{v_c^{3/2}}$$

$$k_c = \frac{a_{c,4} - 4a_{c,3} \cdot a_{c,1} + 6a_{c,2} \cdot a_{c,1}^2 - 3a_{c,1}^4}{v_c^2}$$

$$cov_{c_i,c_j} = b_{i,j} - \mu_{c_i} \cdot \mu_{c_j}$$

Using the CIE $L^*a^*b^*$ color space, we obtain 12 moments and 3 covariances, a total of 15 feature values. We can combine the values into a vector \mathbf{m} (in a defined order) and compare to feature vectors \mathbf{m}_i and \mathbf{m}_j of two images using either Euclidean or Manhattan distance:

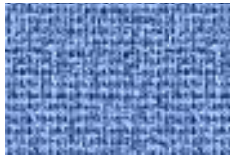
$$\delta_{Manhattan}(\mathbf{m}_i, \mathbf{m}_j) = \sum_{k=1}^{15} |\mathbf{m}_{i,k} - \mathbf{m}_{j,k}|$$

$$\delta_{Euclidean}(\mathbf{m}_i, \mathbf{m}_j) = \sqrt{\sum_{k=1}^{15} (\mathbf{m}_{i,k} - \mathbf{m}_{j,k})^2}$$

- Variants: like with histograms, we can construct moments for brightness/luminance only. Co-variance becomes obsolete and we obtain 4 brightness/luminance moments. We can further construct moments only for the chromaticity aspect, disregarding brightness/luminance. In this case we have 8 moments and one covariance value, resulting in a 9 dimensional feature.
- Discussion:
 - The value ranges of moments vary significantly. Before we can apply a distance measure, we need to scale the values into the same range (e.g., $[0,1]$). Due to the differences in the distance measure, it is sufficient to just scale the values either by $max - min$ of each component, or the standard deviation of the values along this dimension (not to be confused with the variance color moments; the standard deviation is taken from the actual values along each moment). We can obtain this scaling factors from a large enough sample set and use them as constant factors when extracting the features.
 - Color moments, like histograms, are robust against translation, rotation, noise, and scale; in some cases, also against lightning differences. The lack of spatial relation between colors may lead to unexpected results (like with histograms).
 - In contrast to histograms, the color moments are independent from each other and we do not need a cross-correlation matrix for a quadratic distance function. The resulting vectors are also much shorter (15 if all moments are taken) than the histograms (up to 1000 bins possible). The compact representation leads to obvious performance gains but no loss in retrieval quality.

4.6 Texture Information

- Texture describe the structure of a surface or part of the image and provides us with information about the spatial arrangement of colors, changes in this arrangement, and the direction and frequency of these changes. We can analyze texture in three ways:
 - Structural approach: Find sets of primitive so-called texels that are composed to regular and repeated patterns as per the examples below:



This approach is limited to artificially generated images and does not work for natural images. The inverse problem of creating texture on the surface of objects is well supported by today's graphic processors (see texels, and Voronoi tessellation).

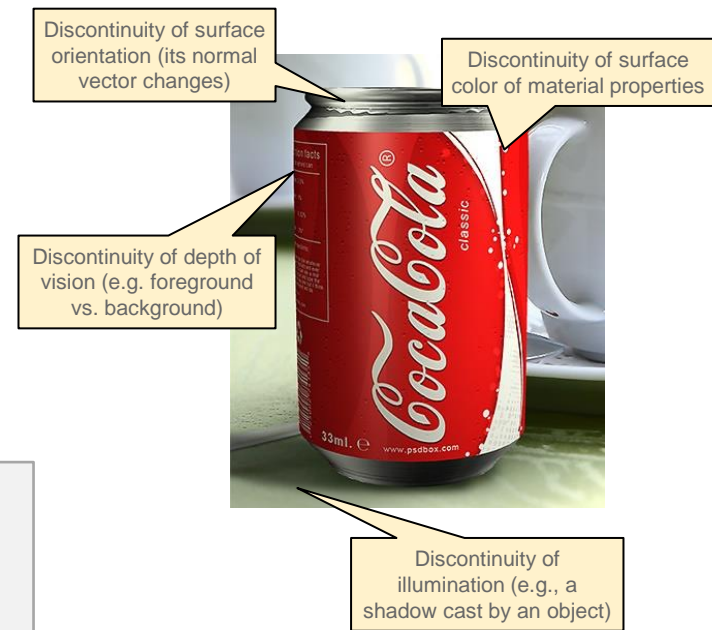
- Statistical approach: Measure the arrangements in the neighborhood of pixels, quantify them, and create statistical summaries (histograms, moments). We will look at edge detection and optimized filters to get texture features.
- Fourier approach: Transform the image into the frequency space via Fourier transformation and extract information about the support for so-called Gabor filters in the frequency space.
- Often, we study texture only in grayscale images. For that purpose, we can compute the Y or L^* components in the CIE color models. Recall, that the original picture first needs to be transformed to linear RGB before computing the transformation to CIE XYZ and CIE $L^*a^*b^*$ (see sRGB \rightarrow linear RGB). In the following, we assume monochromatic images with only a brightness/luminance channel. Advanced methods may also consider chromaticity information for textures.

- **Edge magnitude and direction** (structural approach)

- Edges in images are caused by several factor as shown on the picture on the right hand side. The detection of edges is the search for gradients with high energy (abrupt change of neighboring pixels). The standard approach is to apply a Sobel operator (convolution) on a smoothed (Gaussian) version of the image, and to determine g_x and g_y values for a pixel. The kernel matrices are given as:

$$\mathbf{G}_x = \frac{1}{8} \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix}$$

$$\mathbf{G}_y = \frac{1}{8} \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$



We can omit the factor 1/8 but then the gradient values are 8 times larger (not a problem for the method shown here). The operators yield a g_x and g_y for each pixel. We can now compute the gradient magnitude $g_{mag}(x, y)$ and the direction of the gradient $g_{dir}(x, y)$ as follows:

$$g_{mag}(x, y) = \sqrt{g_x(x, y)^2 + g_y(x, y)^2}$$

$$g_{dir}(x, y) = \arctan(g_x(x, y), g_y(x, y))$$

$\arctan(x, y)$ is the arc tangent of y/x taking the quadrant of (x, y) into account

- With the above transformation, we obtain 2 values for each pixel in the image. The first value describes how large the change is (energy), the second value represents the direction of change (from darker to lighter). A value of $g_{dir} = 0$ is a vertical edge (change direction is normal to the edge) and the lighter pixel is on the right hand side.

- We now can create simple texture based features.
 - Edginess of image: Proportion of image with $g_{mag}(x, y) \geq \tau$ for a given threshold τ . This expresses how many edges we can expect on the picture with high enough energy. Continuous areas of them image with, for example, the sky or a lake will result in low values, while several objects or city images with lead to higher values.

$$f_{edginess} = \frac{1}{N \cdot M} \sum_{x,y} \begin{cases} 1 & \text{if } g_{mag}(x, y) \geq \tau \\ 0 & \text{otherwise} \end{cases}$$

- Gradient Histograms: same approach as with color histogram. We now have to values per pixels and quantify the direction and the magnitude. The distance between reference gradients is calculated similar as for the HSV color model. Recall that differences in direction are calculated as $\min(|\alpha - \beta|, 2\pi - |\alpha - \beta|)$. We need to normalize energy and direction ranges to compute the distance $d_{i,j}$ between two reference gradients. This allows us to compute the matrix **A** for the quadratic distance measure. Given to histograms **h** and **g**, and assuming N reference gradients, we obtain distances as follows:

$$\delta_{Manhattan}(\mathbf{h}, \mathbf{g}) = \sum_{i=1}^N |h_i - g_i|$$

$$\delta_{Euclidean}(\mathbf{h}, \mathbf{g}) = \sqrt{\sum_{i=1}^N (h_i - g_i)^2}$$

$$\delta_{quadratic}(\mathbf{h}, \mathbf{g}) = (\mathbf{h} - \mathbf{g})^T \mathbf{A} (\mathbf{h} - \mathbf{g})$$

$$\mathbf{A}: a_{i,j} = 1 - \frac{d_{i,j}}{\max_{k,l} d_{k,l}}$$

Distance normalized by maximum distance for all pairs of reference gradients

As with color histograms, the same issues with high dimensionality occurs.

- **Gradient Moments:** as before, we compute moments for the magnitude and the direction, and a covariance value for magnitude and direction. Let c denote either magnitude or direction:

$$\begin{aligned}\mu_c &= \frac{1}{N \cdot M} \sum_{x,y} g_c(x,y) & v_c &= \frac{1}{N \cdot M} \sum_{x,y} (g_c(x,y) - \mu_c)^2 \\ s_c &= \frac{1}{N \cdot M} \sum_{x,y} \left(\frac{g_c(x,y) - \mu_c}{\sqrt{v_c}} \right)^3 & k_c &= \frac{1}{N \cdot M} \sum_{x,y} \left(\frac{g_c(x,y) - \mu_c}{\sqrt{v_c}} \right)^4 \\ cov_{mag,dir} &= \frac{1}{N \cdot M} \sum_{x,y} (g_{mag}(x,y) - \mu_{mag}) - (g_{dir}(x,y) - \mu_{dir})\end{aligned}$$

This results in 9 feature values describing the distribution of gradients.

- **Laws' Texture Energy** (structural approach)

- Laws texture masks compute 9 values for a pixel in the image to capture various aspects of texture features. The masks are based on 4 prototype vectors:

$v_{L5} = [1 \quad 4 \quad 6 \quad 4 \quad 1]$	Level: (Gaussian) center-weighted local average
$v_{E5} = [-1 \quad -2 \quad 0 \quad 2 \quad 1]$	Edge: (gradient) responds to step edges
$v_{S5} = [-1 \quad 0 \quad 2 \quad 0 \quad -1]$	Spot: (Laplace of Gaussian) detects a spot
$v_{R5} = [1 \quad -4 \quad 6 \quad -4 \quad 1]$	Ripple: (Gabor) detects ripples

- From these base vectors, we can compute 16 matrices by multiplication of pairs of prototype vectors. For the instance E5L5, for instance, we obtain the Kernel matrix \mathbf{G}_{E5L5} as follows:

$$\mathbf{G}_{E5L5} = \mathbf{v}_{E5}^T \mathbf{v}_{L5} = \begin{bmatrix} -1 \\ -2 \\ 0 \\ 2 \\ 1 \end{bmatrix} [1 \quad 4 \quad 6 \quad 4 \quad 1] = \begin{bmatrix} -1 & -4 & -6 & -4 & -1 \\ -2 & -8 & -12 & -8 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 2 & 8 & 12 & 8 & 2 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

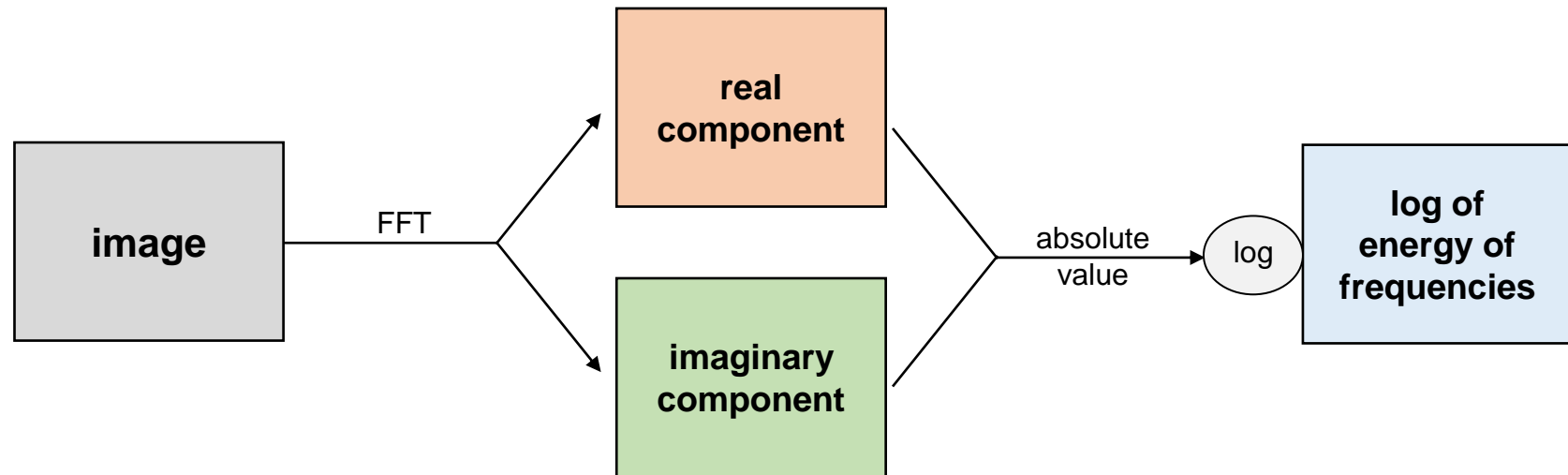
Since E5L5 and L5E5 measure a similar aspects, we collapse them into a single Kernel and use the average of both matrices. With such reductions, we obtain 9 Kernel matrices:

$$\mathbb{G} = \left\{ \frac{\mathbf{G}_{E5L5} + \mathbf{G}_{L5E5}}{2}, \frac{\mathbf{G}_{L5R5} + \mathbf{G}_{R5L5}}{2}, \frac{\mathbf{G}_{E5S5} + \mathbf{G}_{S5E5}}{2}, \frac{\mathbf{G}_{S5L5} + \mathbf{G}_{L5S5}}{2}, \frac{\mathbf{G}_{E5R5} + \mathbf{G}_{R5E5}}{2}, \frac{\mathbf{G}_{S5R5} + \mathbf{G}_{R5S5}}{2} \right\} \\ \cup \{ \mathbf{G}_{S5S5}, \mathbf{G}_{R5R5}, \mathbf{G}_{E5E5} \}$$

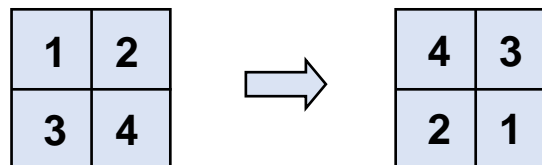
- With these 9 Kernel matrices, we apply a convolution to obtain 9 texture energy values $e_i(x, y)$ per pixel (with $1 \leq i \leq 9$). From here, we can apply the same approaches as before:
 - Histograms: although feasible, we are faced here with 9 values per pixel. If we quantize them with 4 ranges, we obtain $4^9 = 262,144$ reference energies. This clearly exceeds our expectations of a computationally meaningful feature, especially, if we consider the necessity of a quadratic function. Using only 2 ranges yields $2^9 = 512$ reference energies. Acceptable, but the quantification error is significant.
 - Moments: for each energy value, we can calculate 4 moments, and co-variance values for the 36 possible pairs. This yields a 72 dimensional feature vector. If the dimensionality is too high, we can reduce the number of moments (only first 2 or 3) or omit the co-variances.

- **Gabor Moments** (Fourier approach)

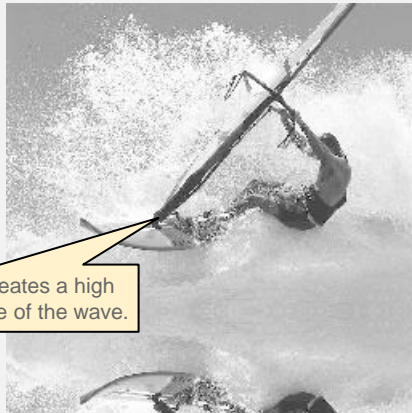
- The 2D Fourier transformation maps a (grayscale) image into its frequency space. More formally, it creates a real and imaginary matrix. For the visualizations, we can compute the log of the sum of squared components (the log-function helps for visualization of the large differences in energy). The 2D Fast Fourier Transformation is an accelerated version of the algorithm reducing computational efforts significantly. However, it is only applicable to image sizes of $2^a \times 2^b$. The picture bellow depicts the transformation:



- To display the frequencies such that low frequencies are in the middle and high frequencies in the outer areas, we need to map the quadrants of the matrix as per below:



- **Examples for the frequency map:** The pictures below show the grayscale original images, and the log-scaled frequency map; the brighter a pixel, the more energy for the corresponding frequency. Low frequencies are in the center, high frequencies in the out areas. The direction from the center to the frequency denotes the normal of an edge in the image for that frequency.



Mast of the sail creates a high contrast to the white of the wave.

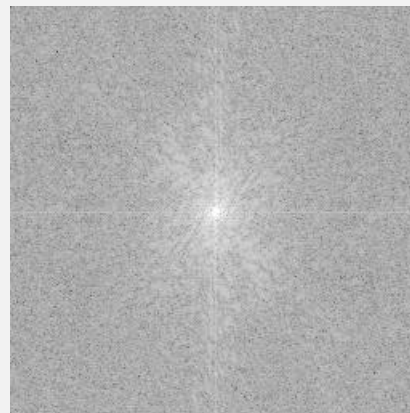
FFT



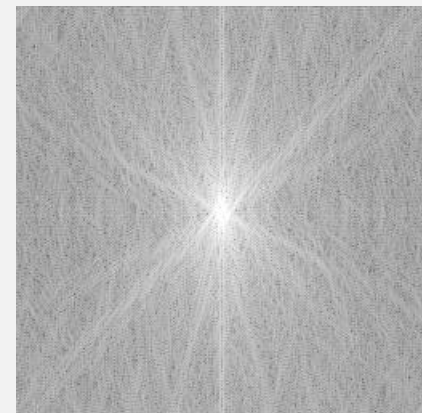
This spike corresponds to the edge of the mast of the sail. The spike is orthogonal to the mast.



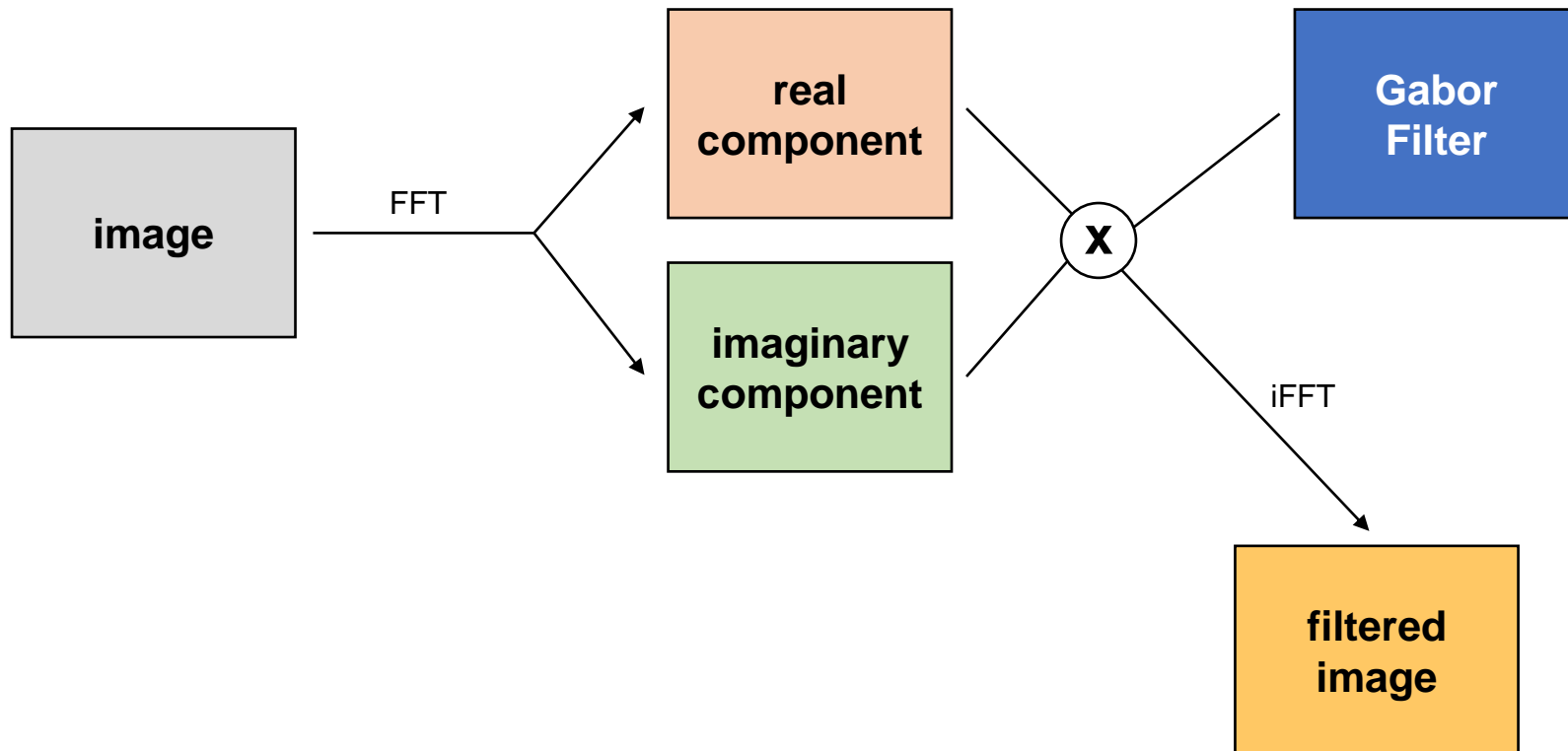
FFT



FFT



- In the Fourier space, we apply a bank of so-called Gabor filters that select different ranges of frequencies and directions. The Gabor filter is multiplied with the Fourier transformation of the image (a complex matrix), and the result is mapped back via inverse Fourier transformation (here the fast implementation iFFT) to the image space. The filtered image now provides information about the support for the selected frequencies and directions in the original image space. Using banks with 5 orientations and 3 scales, we have 15 Gabor filters and obtain 15 different filtered images. We extract statistical moments for each of these filters to obtain a wide range of texture descriptors. The following pages show the filter banks and its application in the Fourier space.



- The Gabor filter is defined as a Gaussian kernel multiplied by a complex sinusoid. In Neurophysiological experiments, it was shown that the Gabor filters, with the right parameters, behave similar to the receptive fields in the primary visual cortex. Its definition is as follows

$$g_{\lambda,\theta,\varphi,\sigma,\gamma}(x,y) = e^{-\frac{\bar{x}^2 + \gamma^2 \bar{y}^2}{2\sigma^2}} \cdot e^{i(2\pi\frac{\bar{x}}{\lambda} + \varphi)}$$

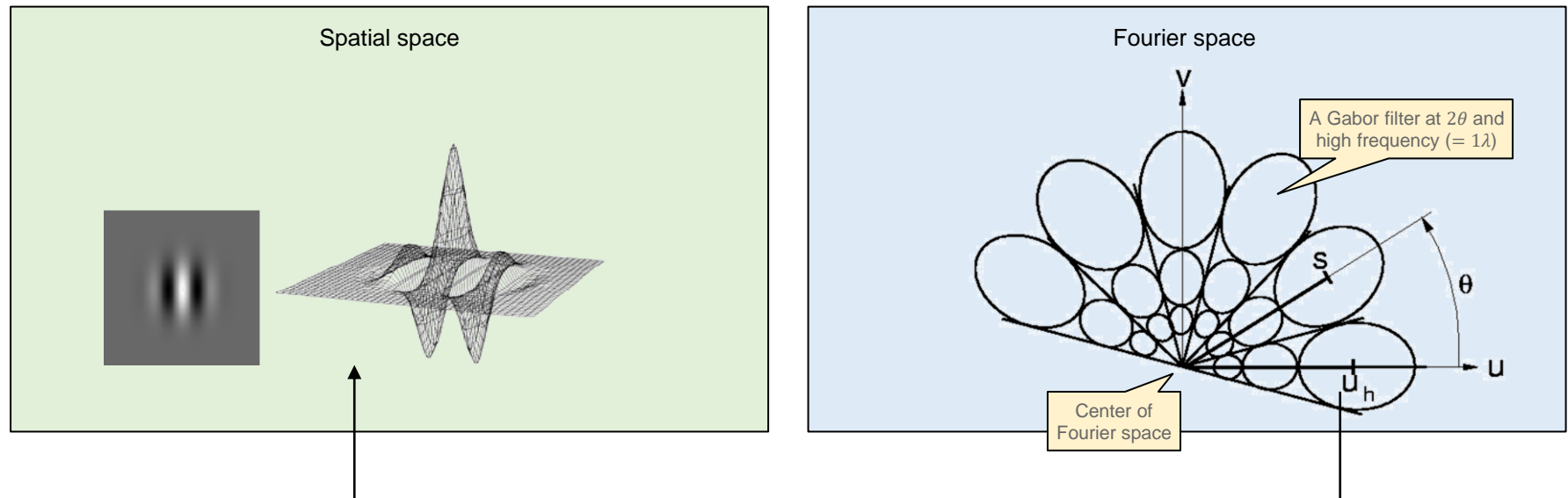
Gaussian kernel with standard deviation σ and the spatial aspect ratio γ

Complex sinusoid with phase φ and wavelength λ . $1/\lambda$ is the frequency of the sinusoid.

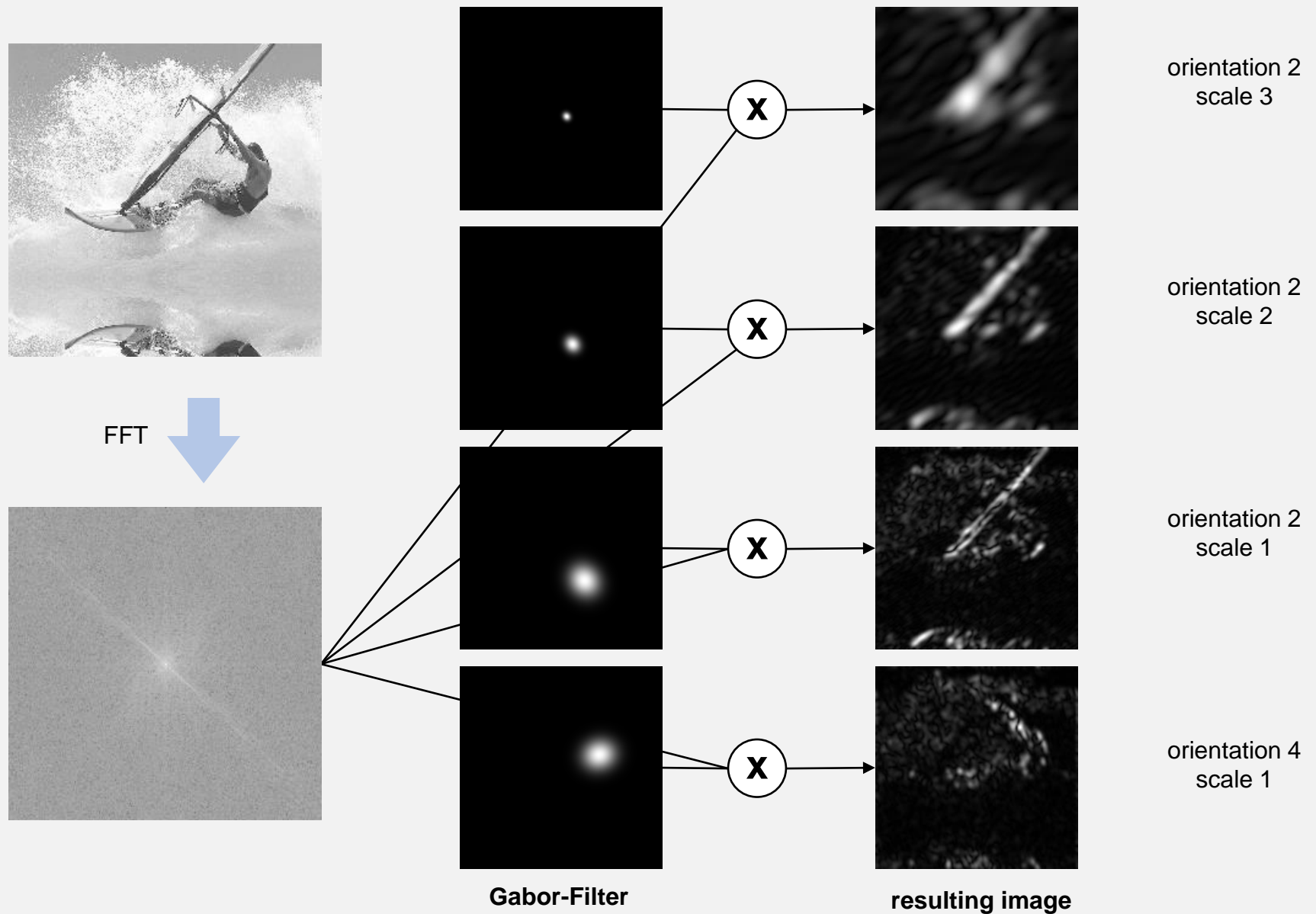
$$\bar{x} = x \cos \theta + y \sin \theta$$

$$\bar{y} = -x \sin \theta + y \cos \theta$$

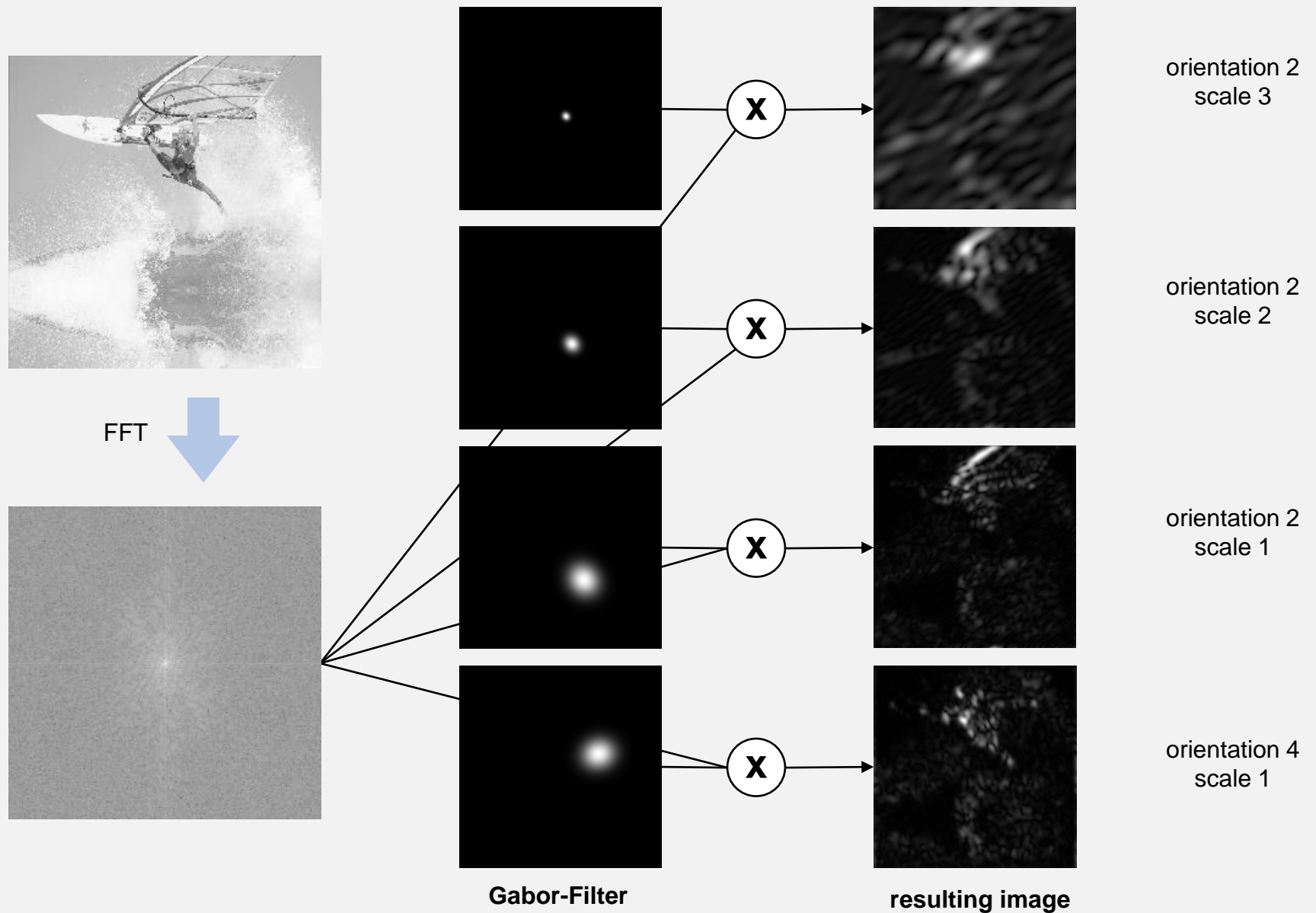
Before application to the Gaussian and sinusoid, the coordinates are rotated by θ . With this definition and varying the parameters, it is possible to construct various filters that are sensitive to frequencies and direction. Mapping the Filter bank into the Fourier space leads to the following layout:



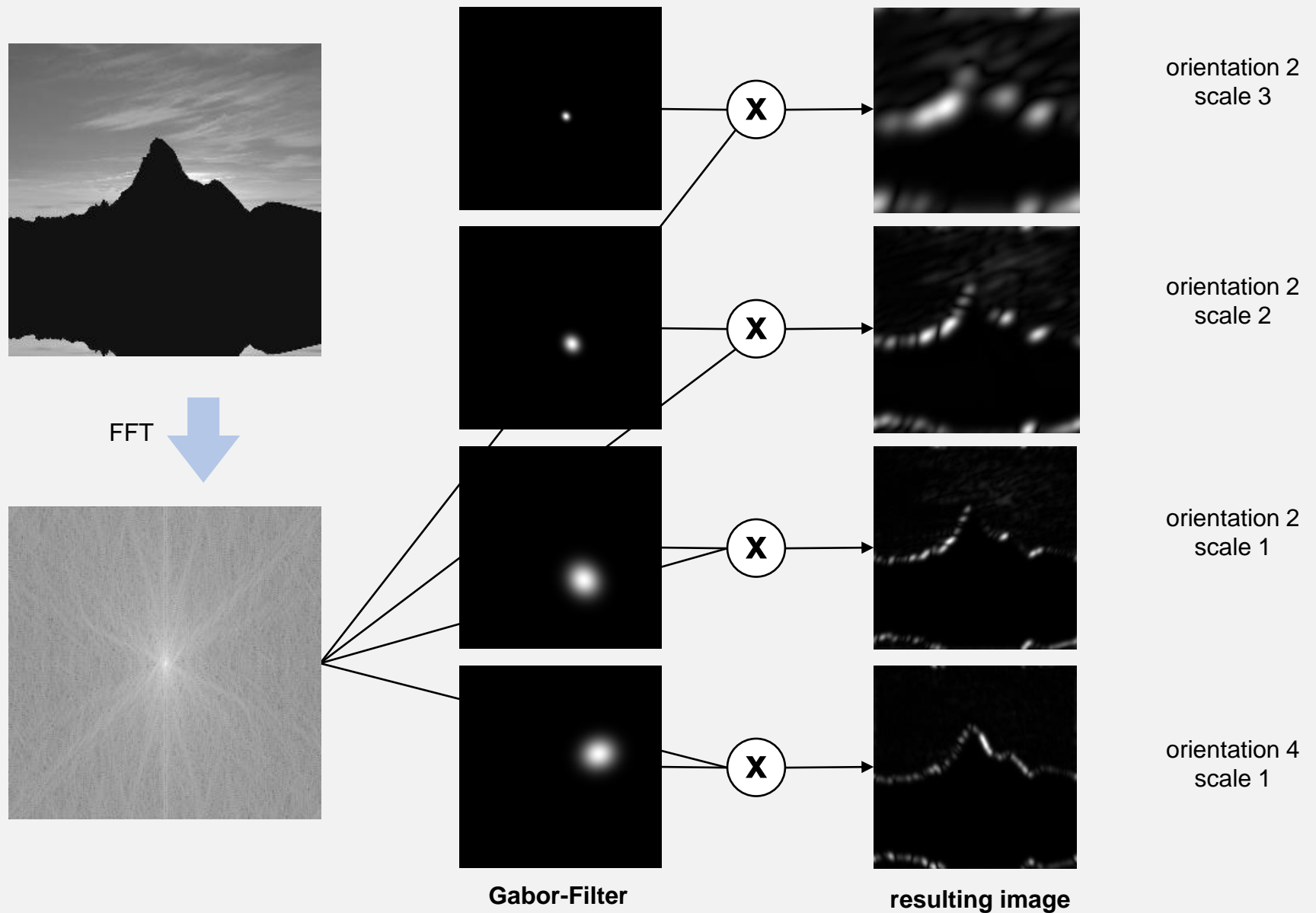
- **Example (1)**



- **Example (2)**



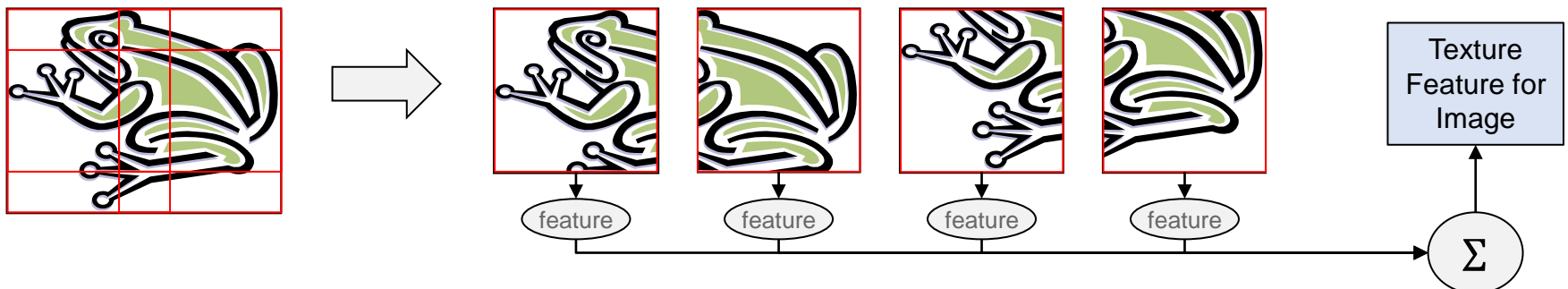
- **Example (3)**



- There are two approaches to compute Gabor filtered images:
 - **Fourier space:** compute the Gabor filters in the Fourier space and apply them to the Fourier transformed image. To enable the use of FFT, the size of the image is scaled to the next higher $2^a \times 2^b$ dimension with one of the following methods
 - **Stretching:** stretch the image to match the new size. This changes proportions and thus frequencies and directions in the image.
 - **Filling:** copy the image 1:1 and fill the remaining area with a neutral color.
 - **Tiling:** create a 2-by-2 tile of the same image and crop to the new size.
 - **Mirroring:** create a 2-by-2 tile, but mirror the image at the middle axis. This reduce hard edges that otherwise become visible as spikes. But it adds wrong directions.



A further alternative: we use the next smaller $2^a \times 2^b$ dimension and apply the method 4 times for the $2^a \times 2^b$ areas in each corner. At the end, we average all feature values across all areas.



- **Image/spatial space:** compute a Gabor filter bank and apply it to the image through convolution. Since the Gabor filter is complex, we take absolute values of the resulting complex numbers to map back to real numbers. Most image processing libraries (OpenCV, scikit-image) provide implementations for Gabor kernels.
- Once we have the filtered images (like shown in the right hand columns on the pages before with the image examples), we can summarize the results with the usual approaches of histograms or moments. We typically select 3-7 directions ($0 \leq \theta \leq \pi$) and 2-5 scales (or frequencies; $1/\lambda$ usually measured in pixels and ranging from 0.05 to 0.5). With a large number of filters, the moments are again a better choice to reduce the number of dimensions and avoid the complexity of quadratic distance functions.
- With moments, we simply treat the absolute values in the filtered image as the raw data points and compute mean, variance, skewness, and Kurtosis on these values. To further reduce the number of dimensions, it is possible to select only the first 2 or 3 moments. Let $\tilde{f}_i(x, y)$ be the filtered (complex) image representation after applying the i -th Gabor filter. We obtain:

$$\mu_i = \frac{1}{N \cdot M} \sum_{x,y} |\tilde{f}_i(x, y)|$$

$$v_i = \frac{1}{N \cdot M} \sum_{x,y} (|\tilde{f}_i(x, y)| - \mu_i)^2$$

$$s_i = \frac{1}{N \cdot M} \sum_{x,y} \left(\frac{|\tilde{f}_i(x, y)| - \mu_i}{\sqrt{v_i}} \right)^3$$

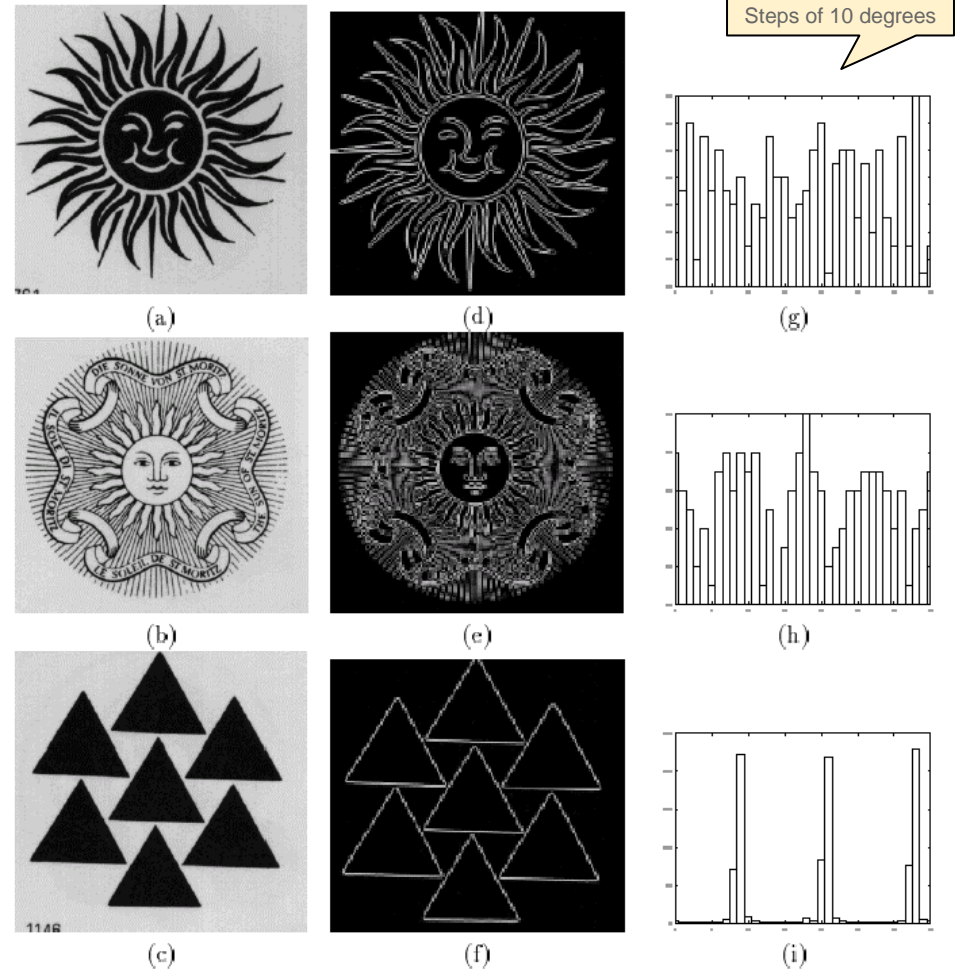
$$k_i = \frac{1}{N \cdot M} \sum_{x,y} \left(\frac{|\tilde{f}_i(x, y)| - \mu_i}{\sqrt{v_i}} \right)^4$$

The overall feature is simply the concatenation of all moments across all filters.

4.7 Shape Information

- In this section, we consider three approaches to define shape features.
 - Identify key shape related features in the entire image. There are no segments or objects taken into account, i.e., the features are global for the image.
 - Given a segmentation of the image into objects/blobs, describe the shape of this region to retrieve similar shape from the database. This also works for 2D/3D objects.
 - Identify key points of interest in the picture and describe these points to identify similar objects. This method is used for stitching of panorama images, object recognition, and motion detection.
- **Global Features:** very similar to the texture features, but we are more interested in the contours and direction of these contours than the rest of the image. The basic idea is to apply an edge detector to obtain the outlines of the principle shapes of the image. The Canny edge detector is a solid reference detector with 5 phases (the first two steps are the same as before with texture):
 1. Apply Gaussian filter to smooth the image and to remove noise or compression artifacts
 2. Compute gradients with their magnitude and direction (as seen before, Sobel operators)
 3. Eliminate values that are not a local maximum in the positive/negative direction of the gradient
 4. Identify strong edges (magnitude above high threshold) and weak edges (magnitude between low and high threshold) and eliminate values below low threshold.
 5. Track edges and eliminate isolated weak edges. Keep only weak edges if in their immediate proximity, there is a strong edge.

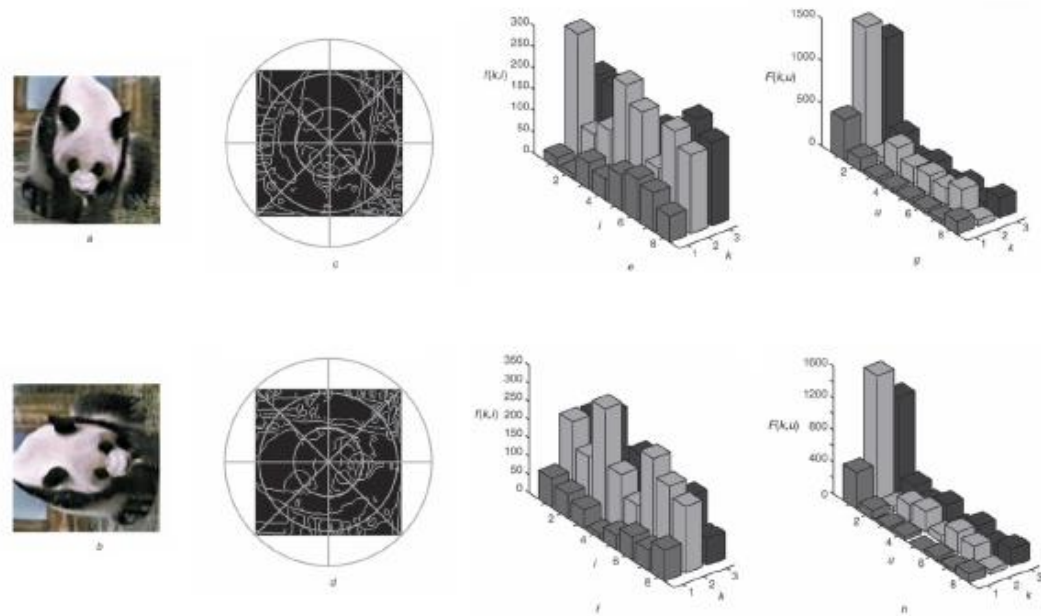
- With the edges, we now can summarize the directions of these edges (the magnitudes have been eliminated in the process) with histograms. The examples on the right side are from an early prototype by Vailaya (1996), Michigan State University.
- The histograms are normalized by the number of edge pixels and sum up to 1. The step size was 10 degrees hence 36 bins for the histograms.
- Comparison between histograms is based on the usual distance function. Again, a quadratic distance function is recommended to account for the similarity between angles
- The feature is translation and scale invariant. With appropriate normalization of the image, we can achieve lightning invariance. However, it is not rotational invariant.
- To obtain rotational invariance, we need to determine the principle direction and rotate the image such that the principle direction points, for example, upwards. The principle direction is the weighted sum of the original gradients, with the magnitude as weights.



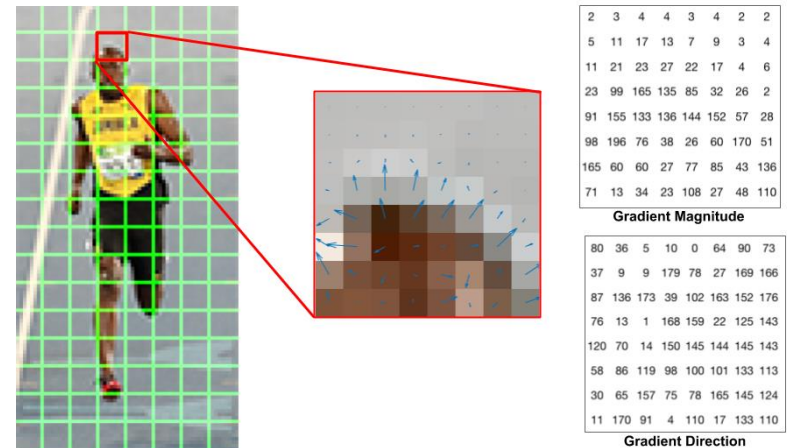
- The **Angular Radial Partitioning** by Chalechale (2003) follows a similar approach to detect edges but uses a different approach to create histograms. The method has 5 steps
 1. Convert the images to grayscale, e.g., by mapping pixels to the L^* -channel
 2. Normalize size of images to obtain comparable numbers
 3. Apply Canny edge detector to find strong edges in the image
 4. Partition the resulting edge-map into $M \times N$ radial angular partitions. M is the number of radial sectors, N the number of slices
 5. Count the number of edge pixels in each partition to obtain a raw histogram
 6. Apply a Fourier transform to the histogram and use absolute values (energy) to obtain the final feature vector

The method is depicted on the right side with an example from the paper.

- The feature is robust against translation and scale due to initial normalization process. It is robust to small rotational changes as only few pixels will change the partition.
- The feature is robust against discrete rotations of the angle of the slice due to the Fourier transformation.
- The feature is robust against omissions of smaller details and noise during edge detection.



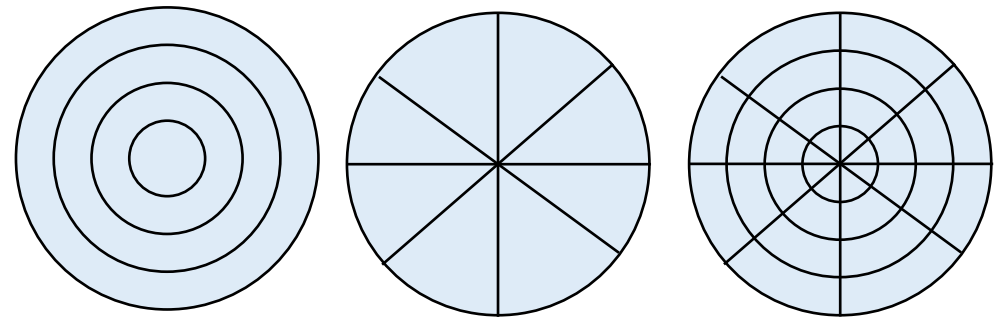
- The **Histogram of oriented gradients** dates back to 1986 but regained interest with the work of Dalal and Triggs in 2005 to detect pedestrians. The methods has since been extended and is often used as input into neural networks.
 - Step 1: compute gradients, for instance, with Sobel operators on a grayscale version of the image. In contrast to other approaches, HOG uses unsigned gradients, i.e., the direction lies in the range of 0 to π . Values between π and 2π are rotated by π . Some HOG implementation let users choose between unsigned and signed gradients, but Dalal and Triggs found that this worked best for pedestrian detection
 - Step 2: As shown in the picture below, the image is divided into cells each with 8x8 pixels. For each of the cell, HOG computes a 9-bin histogram (9 was found to be optimal for their use case) over the gradient directions of the 64 pixels and weighted by their gradient magnitudes.
 - Step 3: gradient magnitudes are variant to illumination and hence require normalization before we can compare histograms with each. Rather than normalizing the 9-bin histograms at each cell, HOG combines 4 neighboring cells and normalizes the concatenated histograms (now 36 bins) so it sums up to 1. The 4 neighboring cells (2x2 cells, each with 8x8 pixels) are moved along the image in steps of 8 pixels. Each block yields a normalized histogram of 36 bins. These blocks are partially overlapping.
 - Step 4: combine histograms to global features or keep a “bag” of local features for search.
 - Optional: The HOG features can be used as input into machine learning algorithm. Dalal and Triggs used an SVM to detect pedestrians.



- **Descriptions of blobs/regions/objects:** given a set of segments, blobs or objects, we can describe the regions based on a set of simple spatial metrics. Due to different resolutions and the absence of a standard size of a pixel (unless provided by the image format), spatial metrics are often in relation to the entire image. For example:
 - Area: percentage of pixels within the segment (over the entire image)
 - Centroid: average of all x-values and of all y-values in the region (in absence of mass values)
 - Axis of Least Inertia: this is the axis which allows the rotation of the object with least energy. It is given by the line that minimizes the squared distances to the boundary of the region. This can be used to normalize regions into a primary direction
 - Eccentricity: given a bounding box in the principle direction, the ratio of length to width of the box denotes the eccentricity
 - Circularity Ratio: how closely the shape resembles a circle. There are different definitions, for instance, the ratio of the area of the smallest circle containing the region to the area of the region
 - ...and many more

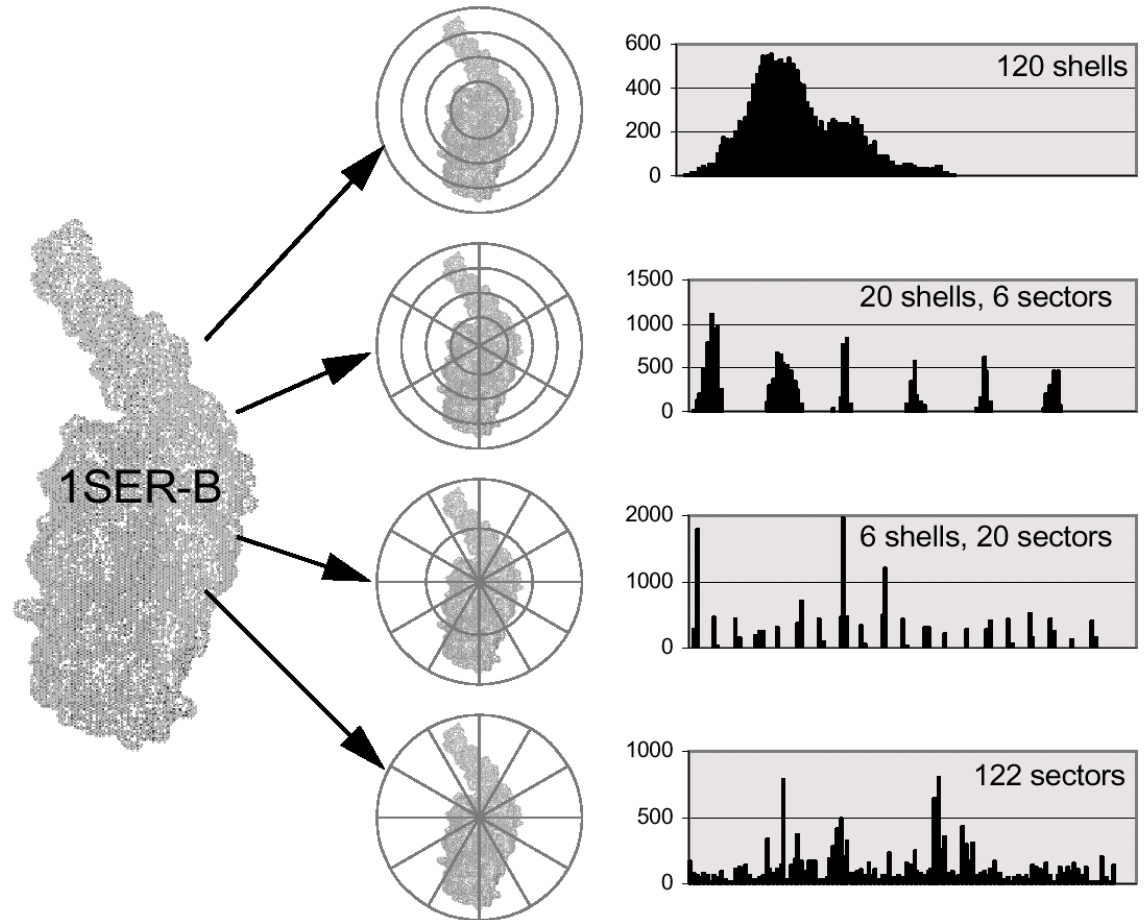
An alternative approach is to normalize the position of the region (principle direction points upwards) and to measure the overlap with a predefined grid to compute histograms. The histogram values are the relative area covered by the grid. There are different ways to define the grid, for instance:

- The grid is always such that it contains the region and is as small as possible.
- With the circular structures, the center is the center of gravity, and the radius is the largest distance of a point to the center of gravity.

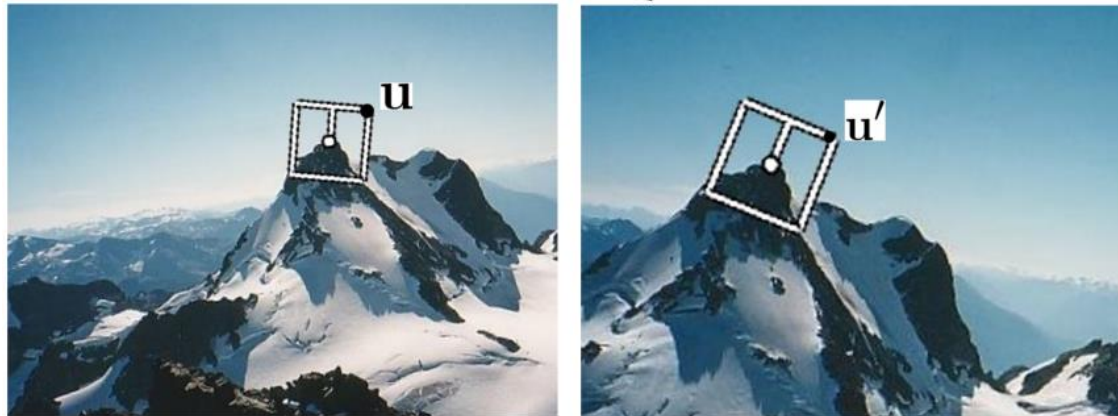


- **Ludwig-Maximilians University Munich** (Berchtold, 1997) studied methods to compare and index 2D and 3D objects. But the methods are similarly applicable to recognized segments in an image. The example on the right side shows 2 complex molecule structure normalized in direction. The partitioning methods extract 4 different histograms, each with 120-122 bins. This is the description of the structure and can be used in combination with a distance measure to find similar objects in the database

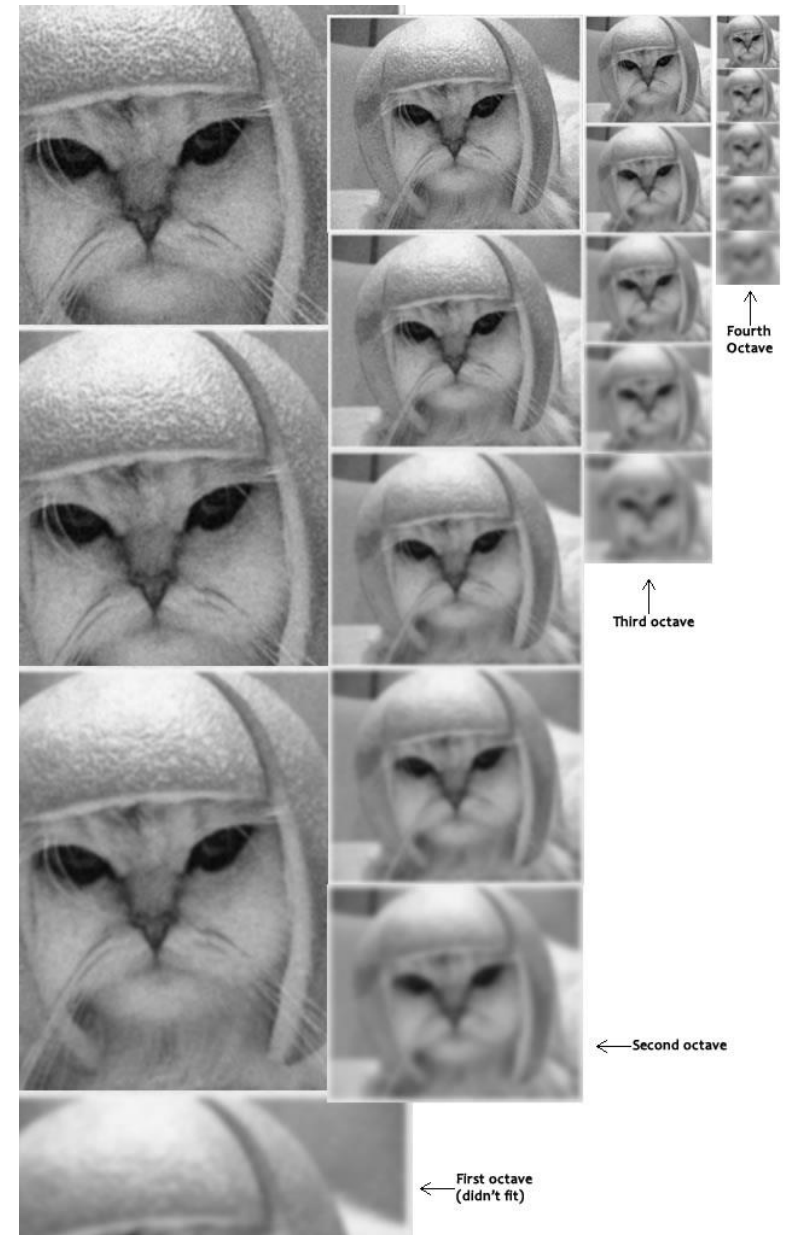
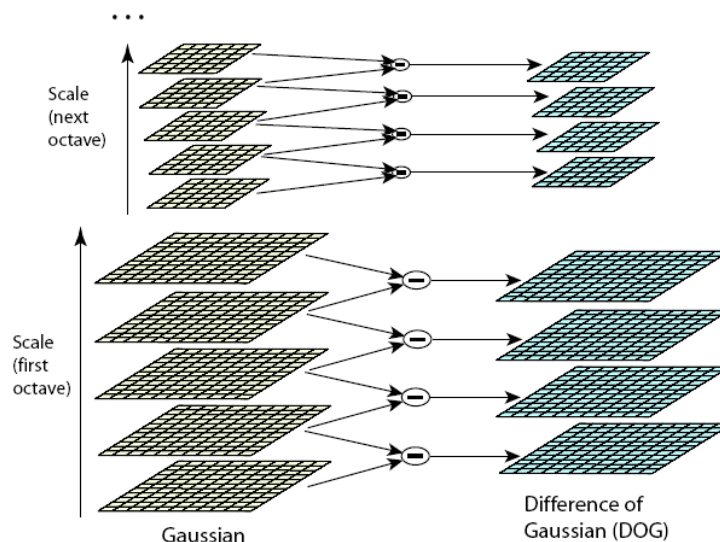
- To make the feature scale invariant, the histogram bins are normalized to sum up to 1.
- Some of the features are rotation invariant (like the first partitioning). With the initial normalization to a principle direction, rotation invariance is given for all partitioning scheme.
- The feature is translation invariant due to the use of the center of gravity.



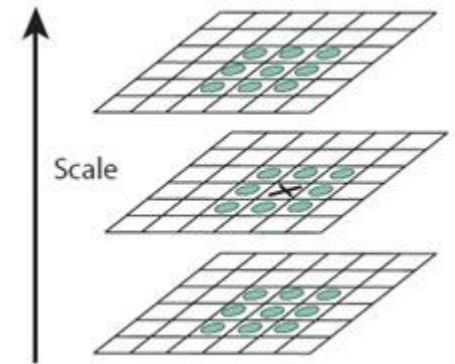
- **Key Points of Interests:** There are many approaches but we consider here only the **Scale Invariant Feature Transform (SIFT)**. Due to the complexity of the approach, we summarize the main steps to identify key points of interest and consider how to describe these points to find matches. SIFT extracts features in a very robust way, so that they match again even after significant viewpoint changes. SIFT is used for object recognition, image stitching, motion tracking, and many other use cases, The images below depict the same mountain from slightly different perspective. SIFT is able to match the two highlighted key points despite rotation and scale differences.
 - The algorithms works roughly in 4 steps
 1. Identify scale-space extrema using band-pass filters (difference of Gaussians, DOG)
 2. Keypoint localization with scale; these are the resulting points of interest
 3. Orientation assignment (primary direction of the region around a keypoint for normalization)
 4. Keypoint descriptors that can be used for similarity search



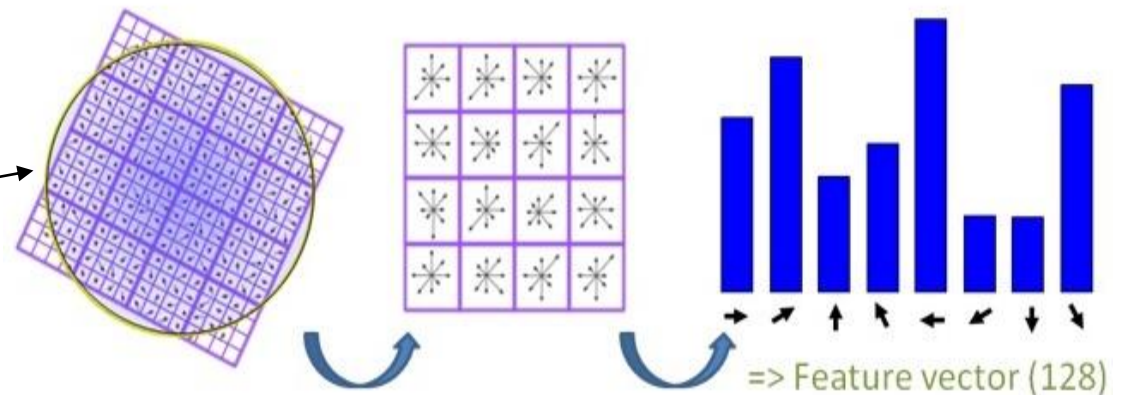
- Step 1: We create a pyramid of images using Gaussian filters at different standard deviations σ and scales. SIFT calls the different scales “octaves” as shown on the right side. Each octave is down sampled to a $\frac{1}{4}$ of the previous octave. For each octave, the image is progressively blurred (Gaussian filters with increasing σ).
- In each octave, neighboring images are subtracted to create the difference of Gaussians (DOG) which act like edge detectors for a defined frequency band
- The DOG image pyramid contains potential edges and point of interests. They are the local minima and maxima in the DOG.



- Step 2: We detect the local minima and maxima in the DOG pyramid with a one pixel neighborhood. As shown in the picture on the right, where “x” marks the current pixel, we have 8 neighbors in the same plane and 9 neighbors from each the plane above and below. If the pixel is a maxima or minima in this neighborhood, mark it as such. Otherwise dismiss the pixel.
 - Starting with 5 Gaussian blurred images in each octave, we created 4 DOG images which now create 2 extrema images at each octave.
 - To thin out the number of keypoints, we dismiss all pixels whose value in the DOG is smaller than a threshold (these are points in the “flat”). We further dismiss all edges by considering their gradients. An edge has big gradient orthogonal to the edge, and a small gradient along the edge. But we are interested in corner points with two big gradients.
 - The output of step 2 is a set of keypoints with location and scale.
- Step 3: To construct a rotation invariant feature, we need to calculate a major orientation for the keypoint. SIFT accumulates a local histogram of gradient directions from the neighborhood of the keypoint. The area of the neighborhood window is proportional to the scale. A gradient direction is added to the histogram with its magnitude as the weight. Finally, the histogram bin with the highest value corresponds to the dominant direction (if there are ties, use all directions).
 - SIFT uses the dominant direction to normalize feature gathering as shown in the next step. If several directions are found, it constructs features for all directions. The normalization allows us to compare keypoints found from different viewpoints with a simple metric.
 - The dominant direction of the keypoint is not necessarily its gradient direction.



- Step 4: Using the keypoint as the center, SIFT lies a 4x4 grid in the dominant direction over the image with the size of the grid being dependent on the scale of the keypoint. For each grid cell, a finer 4x4 mesh defines its neighborhood and a histogram with 8 directions captures the directions within the cell. For each point in this finer mesh, we calculate the gradient orientation and the magnitude. We use the magnitude and a Gaussian weight (based on the distance to the keypoint) to add the direction to the histogram. For each cell of the bigger 4x4 grid we obtain a histogram with 8 values, resulting in a total of 128 feature values.
- The SIFT features are invariant to scale, translation and rotation by construction. It follows the idea of the receptive fields in the primary visual cortex to capture local features based on directions. The features are very distinct for the objects and even small objects can yield many descriptors. Although rather complex in construction, features can be obtained close to real-time. SIFT features are widely used for object recognition, motion detection, image alignment and stitching. OpenCV has a SIFT implementation, scikit-image supports similar approaches (daisy, harris). As with HOG, SIFT descriptors can be used as input for machine learning.



4.8 Blob Recognition (unsupervised clustering)

- With unsupervised learning tasks, the machine learning algorithm observes data set without targets and infers a function that captures the inherent structure and/or distribution of the data. In a clustering scenario, that function is a set of clusters and the ability to assign new data items to one (or several) of the clusters. In this chapter, we study the **k-means clustering** and the **Expectation Maximization over a Gaussian mixture** to infer a mapping of features to clusters. In the context of multimedia data, typical applications are:
 - Feature quantization, i.e., reducing a multivariate feature to a small number of discrete values. The quantized value serve as an approximated or smoothed version of the original ones much like histograms approximates the distribution of data values
 - Cluster analysis, i.e., the validation of the cluster hypothesis and the extraction of clusters to infer labels for the clusters.
 - Image segmentation, i.e., the extraction of different areas in an image that “belong” to each other. In a first step, clustering reduces the number of features through quantization. In a second step, morphological operators build coherent regions for segmentation.
- As we do not know the number of clusters that are present in the data (we have no labels!), we need to guide clustering algorithms in the selection of the optimal number K of clusters. Again, poor choice for the number of clusters can lead to underfitting (extreme case is $K = 1$) and overfitting (extreme case is $K = N$ with N being the number of training items). As we have no targets, we cannot use a validation set to measure accuracy of prediction. Instead, we utilize a target function for the compactness of the clusters and the separation between clusters and must prevent, at the same time, an excessive number of clusters.
- We conclude this section with an example from image segmentation and a very early application called Blobworld.

- **k-means clustering** goes back to the 1960s as an approach to quantify vectors for signal processing. It subsequently became very popular in data mining for cluster analysis. k-means clusters the data set into k clusters in such a way that each data point belongs to the cluster with the nearest centroid (or prototype of the cluster). The centroids are the mean position over all points in the cluster. The centroids divide the space into Voronoi diagrams defining the cluster shapes.
 - Although the computation of the optimal K centroids is a NP-hard problem, there are very efficient heuristics that lead to a (local) optimum. We will first describe the classical approach using Lloyd's algorithm and then re-interpret the approach with Expectation Maximization.
 - Let N be the number of data items with the d -dimensional representations x_1, \dots, x_N . We then want to partition the data items into K sets $\mathbb{S} = \{\mathbb{S}_1, \dots, \mathbb{S}_K\}$ such that the **within-cluster sum of squares (WCSS)**, also called the variance) become minimal, i.e.:

$$\mathbb{S}^* = \operatorname{argmin}_{\mathbb{S}} \sum_{k=1}^K \sum_{x \in \mathbb{S}_k} \|x - \mu_k\|_2^2 = \operatorname{argmin}_{\mathbb{S}} \sum_{k=1}^K |\mathbb{S}_k| \cdot \sigma_k^2$$

with μ_k denoting the mean of items in \mathbb{S}_k , and σ_k^2 being the variance of items in \mathbb{S}_k . With Lloyd's algorithm, we obtain a local optimum with a simple iterative algorithm:

1. Select an initial set of centroids $\mu_1^{(0)}, \dots, \mu_K^{(0)}$ (see later how to select)
2. Assign each data point x to the set $\mathbb{S}_k^{(t)}$ if it is closest to μ_k , i.e., $\|x - \mu_k^{(t)}\| \leq \|x - \mu_l^{(t)}\| \quad \forall l: 1 \leq l \leq K$ (if several centroids are closest, pick one randomly)
3. Calculate the new centroids for the next iteration ($t + 1$):

$$\mu_k^{(t+1)} = \frac{1}{|\mathbb{S}_k^{(t)}|} \sum_{x \in \mathbb{S}_k^{(t)}} x$$

4. Repeat steps 2 and 3 until algorithm has converged

- Initial choice of centroids
 - Random points: pick K random items from the data set. This leads to a spread of centroids across the data space.
 - Random partition: assign each data item to a random cluster (1 to K) and compute centroids over these random clusters. These centroids tend to be closer together near the center of the data set.
 - k-means++: the first centroid is chosen randomly from the data set. Each subsequent centroid (up to K) is chosen from the remaining items with probabilities proportional to the their squared distance to closest centroid. Although more expensive, it leads to much smaller final errors and faster convergence during the iterative part.
- **Expectation Maximization (EM)** (and interpretation of k-means algorithm)
 - Expectation maximization is an iterative method to estimate parameters in a statistical model than cannot be solved in closed form. It assumes that the observations (here: the training set) are obtained from probability distribution, typically a mixture of several distributions with a soft assignment. In k-means, we used a hard assignment, that is, every data point is assigned to exactly one cluster. In EM, soft assignment denotes that cluster assignment of a point follows a conditional distribution. Finally, the objective is to find the soft assignment and the parameters of the distributions (e.g., with Gaussian, these are the means and variances) that best explain the observations (maximum likelihood).
 - Solving above objective function in closed form is not always possible. The EM algorithm consists of two steps: in the expectation step, the distribution parameters are constant and we compute the best soft assignment. In the maximization step, we keep the soft assignment constant and choose the parameters that maximize the objective function. With each step, the objective function increases and eventually converges, but not necessarily to a global maximum.

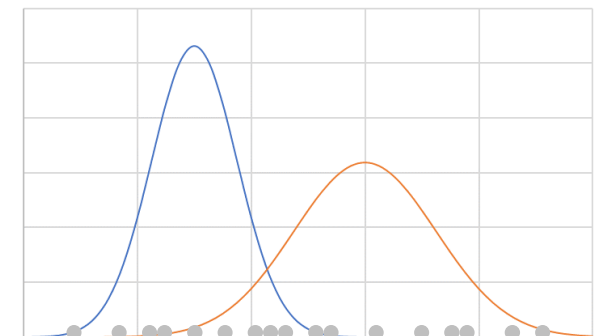
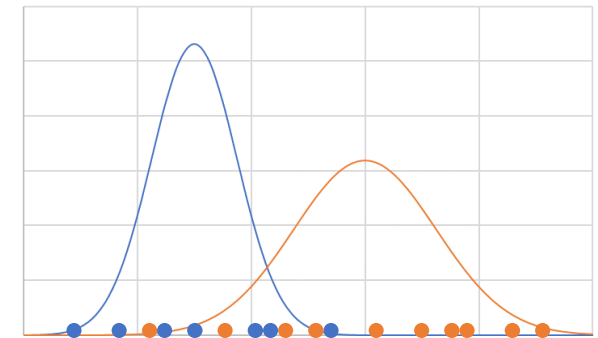
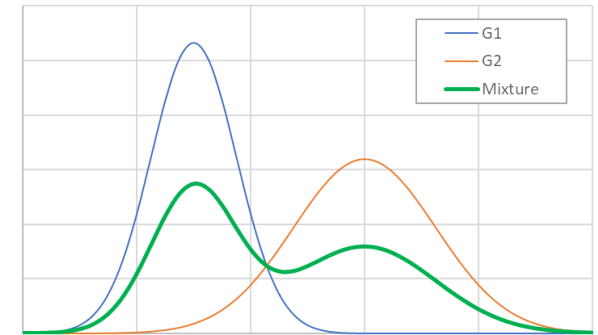
- Let us start with a simple one dimensional example with a mixture of two ($K = 2$) Gaussian distributions $\mathcal{N}(\mu_k, \sigma_k^2)$. The picture on the right shows the two Gaussian distributions and their mixture. With an infinite number of Gaussians, a mixture can model any distribution. Each Gaussian represent a sub-population (cluster) of the data items that follow its distribution. In addition, a prior $P(C_k)$ defines how likely data items come from k -the cluster with $\sum P(C_k) = 1$.
- Now, assume we make the observations $\mathbb{T} = \{x_1, \dots, x_N\}$. Further assume, we know that all $x \in \mathbb{S}_1$ stem from the blue cluster C_1 , and all $x \in \mathbb{S}_2 = \mathbb{T} \setminus \mathbb{S}_1$ stem from the red cluster C_2 . We then can easily compute the parameters and the priors of the distributions using the (biased) estimators:

$$\mu_k = \frac{\sum_{x \in \mathbb{S}_k} x}{|\mathbb{S}_k|} \quad \sigma_k^2 = \frac{\sum_{x \in \mathbb{S}_k} (x - \mu_k)^2}{|\mathbb{S}_k|} \quad P(C_k) = \frac{|\mathbb{S}_k|}{N}$$

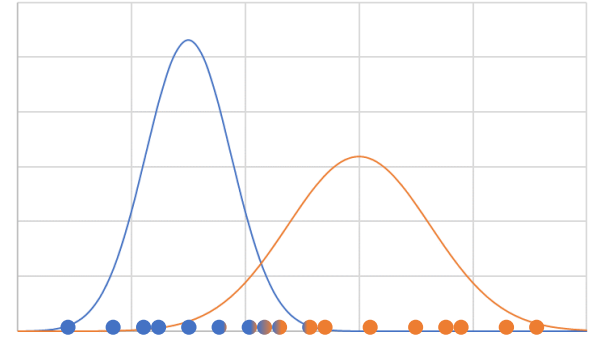
- On the other side, assume we know the parameters μ_k, σ_k^2 of the distributions and the priors $P(C_k)$, can we estimate the probability $P(C_k|x_i)$ that a point x_i is part of cluster C_k ?

$$P(C_k|x_i) = \frac{P(x_i|C_k) \cdot P(C_k)}{P(x_i)} = \frac{P(x_i|C_k) \cdot P(C_k)}{\sum_k P(x_i|C_k) \cdot P(C_k)}$$

with $P(x_i|C_k) = f(x_i; \mu_k, \sigma_k^2) = \frac{1}{\sqrt{2\pi\sigma_k^2}} \cdot \exp\left(-\frac{(x_i - \mu_k)^2}{2\sigma_k^2}\right)$



- Given the probabilities $P(C_k|x_i)$ that x_i belongs to cluster C_k we no longer have a hard assignment as above with $\mathbb{T} = \mathbb{S}_1 \cup \mathbb{S}_2$, and $\mathbb{S}_1 \cap \mathbb{S}_2 = \emptyset$, but utilize soft assignments. In other words, we are not entirely sure from which sub-population the points come from but have a fairly good understanding how likely they stem from each cluster. To estimate the parameters and the priors, we need to take the soft assignments into account:



$$\mu_k = \frac{\sum_i P(C_k|x_i) \cdot x_i}{\sum_i P(C_k|x_i)}$$

$$\sigma_k^2 = \frac{\sum_i P(C_k|x_i) \cdot (x_i - \mu_k)^2}{\sum_i P(C_k|x_i)}$$

$$P(C_k) = \frac{\sum_i P(C_k|x_i)}{N}$$

- Now we can summarize the EM algorithm: to this end, we introduce the **responsibility** $\gamma_{i,k} = P(C_k|x_i)$ denoting the soft assignment of data item x_i to cluster C_k , and the **weights** $w_k = P(C_k)$ representing the prior of cluster C_k . The algorithm runs as follows:

1. Select initial values for $\mu_k^{(0)}$, $\sigma_k^{2(0)}$ and $w_k^{(0)}$ for $1 \leq k \leq K$
2. E-step: evaluate new responsibilities $\gamma_{i,k}^{(t)}$ for $1 \leq i \leq N$ and $1 \leq k \leq K$ using current parameters
$$\gamma_{i,k}^{(t)} = \frac{w_k^{(t)} \cdot f(x_i; \mu_k^{(t)}, \sigma_k^{2(t)})}{\sum_k w_k^{(t)} \cdot f(x_i; \mu_k^{(t)}, \sigma_k^{2(t)})}$$
3. M-step: evaluate new parameters $\mu_k^{(t+1)}$, $\sigma_k^{2(t+1)}$ and $w_k^{(t+1)}$ for $1 \leq k \leq K$ using current responsibilities
$$\mu_k^{(t+1)} = \frac{\sum_i \gamma_{i,k}^{(t)} \cdot x_i}{\sum_i \gamma_{i,k}^{(t)}} \quad \sigma_k^{(t+1)} = \frac{\sum_i \gamma_{i,k}^{(t)} \cdot (x_i - \mu_k^{(t+1)})^2}{\sum_i \gamma_{i,k}^{(t)}} \quad w_k^{(t+1)} = \frac{\sum_i \gamma_{i,k}^{(t)}}{N}$$
4. Repeat E-step and M-step until the parameters stop changing

- Once convergence of EM is reached after ϑ iterations, we can (hard) assign a data item x_i to its most likely cluster C_{k^*} by solving the following equation:

$$k^* = \operatorname{argmax}_k P(C_k | x_i) = \operatorname{argmax}_k \frac{P(x_i | C_k) \cdot P(C_k)}{P(x_i)} = \operatorname{argmax}_k \left(w_k^{(\vartheta)} \cdot f(x_i; \mu_k^{(\vartheta)}, \sigma_k^{2(\vartheta)}) \right)$$

- We can generalize this approach to d -dimensional spaces with $d = M$ being the number of features. We create a mixture of K multi-variate (or multi-dimensional) Gaussian distribution $\mathcal{N}(\mu_k, \Sigma_k)$ with $\mu_k = E[x \in \mathbb{T}_k]$ denoting the centroid of items of cluster C_k , and $\Sigma_k = E_{x \in \mathbb{T}_k}[(x - \mu)(x - \mu)^T]$ the covariance matrix of items in cluster C_k .

1. Select initial values for $\mu_k^{(0)}, \Sigma_k^{2(0)}$ and $w_k^{(0)}$ for $1 \leq k \leq K$
2. E-step: evaluate new responsibilities $\gamma_{i,k}^{(t)}$ for $1 \leq i \leq N$ and $1 \leq k \leq K$ using current parameters

$$\gamma_{i,k}^{(t)} = \frac{w_k^{(t)} \cdot f(x_i; \mu_k^{(t)}, \Sigma_k^{2(t)})}{\sum_k w_k^{(t)} \cdot f(x_i; \mu_k^{(t)}, \Sigma_k^{2(t)})}$$

3. M-step: evaluate new parameters $\mu_k^{(t+1)}, \Sigma_k^{2(t+1)}$ and $w_k^{(t+1)}$ for $1 \leq k \leq K$ using current responsibilities

$$\mu_k^{(t+1)} = \frac{\sum_i \gamma_{i,k}^{(t)} \cdot x_i}{\sum_i \gamma_{i,k}^{(t)}} \quad \Sigma_k^{(t+1)} = \frac{\sum_i \gamma_{i,k}^{(t)} \cdot (x_i - \mu_k^{(t+1)}) (x_i - \mu_k^{(t+1)})^T}{\sum_i \gamma_{i,k}^{(t)}} \quad w_k^{(t+1)} = \frac{\sum_i \gamma_{i,k}^{(t)}}{N}$$

4. Repeat E-step and M-step until the parameters stop changing

- Again, we obtain a hard assignment for a data item x_i to its most likely cluster C_{k^*} as follows:

$$k^* = \operatorname{argmax}_k \left(w_k^{(\vartheta)} \cdot f(x_i; \mu_k^{(\vartheta)}, \Sigma_k^{2(\vartheta)}) \right) \quad f(x_i; \mu_k, \Sigma_k^2) = \frac{1}{\sqrt{(2\pi)^d |\Sigma|}} \cdot \exp \left(-\frac{1}{2} (x_i - \mu_k)^T \Sigma_k^{-1} (x_i - \mu_k) \right)$$

- Where does the name Expectation Maximization come from? Let $\mathbb{X} = \{x_i\}$ be the set of data items and $\mathbb{Y} = \{w_1, \mu_1, \sigma_1, \dots, w_K, \mu_K, \sigma_K\}$ be the set of unknown parameters of the mixture of K Gaussian distributions. In addition, we have the latent unobserved data items $\mathbb{Z} = \{\gamma_{i,k}\}$ denoting the soft memberships of x_i to cluster C_k . Given, \mathbb{X} we want to find the parameters \mathbb{Y} that maximize the probability that the data items in \mathbb{X} are observations from the mixture using these parameters. This is called the **maximum likelihood estimate (MLE)**:

$$\mathbb{Y}^* = \underset{\mathbb{Y}}{\operatorname{argmax}} p(\mathbb{X}|\mathbb{Y}) = \int_{\mathbb{Z}} p(\mathbb{X}, \mathbb{Z}|\mathbb{Y}) d\mathbb{Z}$$

In other words, if \mathbb{Y} is known, how likely is it that data items in \mathbb{X} follow the mixture of the K Gaussian distributions. Adding the soft memberships \mathbb{Z} , $p(\mathbb{X}|\mathbb{Y})$ is given by the marginal probability of $p(\mathbb{X}, \mathbb{Z}|\mathbb{Y})$ over all possible sets of \mathbb{Z} . This equation, however, is often not solvable in closed forms. Instead, an iterative method is used, that improves $\log p(\mathbb{X}|\mathbb{Y})$ with each iteration. EM uses a so-called Q-function that indirectly improves $\log p(\mathbb{X}|\mathbb{Y})$ given current estimates $\mathbb{Y}^{(t)}$:

$$Q(\mathbb{Y}|\mathbb{Y}^{(t)}) = E_{\mathbb{Z}|\mathbb{X}, \mathbb{Y}^{(t)}} [\log p(\mathbb{X}, \mathbb{Z}|\mathbb{Y})]$$

The right hand side is the expectation function over $\log p(\mathbb{X}, \mathbb{Z}|\mathbb{Y})$ given the conditional distribution of \mathbb{Z} given \mathbb{X} and the current estimates $\mathbb{Y}^{(t)}$. Now, the E-step generates this expectation function by computing the probabilities $P(C_k|x_i)$ for \mathbb{Z} (soft assignment) given \mathbb{X} and the current estimates $\mathbb{Y}^{(t)}$ and uses Bayes' rule as we have done above. Then, given \mathbb{Z} , the M-step maximizes the Q-function over all possible \mathbb{Y} to obtain a new estimate $\mathbb{Y}^{(t+1)}$. With log-probabilities and Gaussian distributions, we can cancel log and exp in the equation, and solutions are found by solving for the maximum (partial derivative is zero). We omit proof for solutions and convergence.

- Let us reconsider the k-means algorithm as an EM problem. We can re-write the objective function (within-cluster sum of squares, WCSS) as follows:

$$J = \sum_{i=1}^N \sum_{j=1}^k \gamma_{i,k} \|x_i - \mu_k\|_2^2$$

$\gamma_{i,k}$ are the hard assignments of x_i to C_k , i.e., for each $1 \leq i \leq N$ exactly one $\gamma_{i,k} = 1$ and all others are 0. We can transform k-means to an EM algorithm over a mixture of K Gaussian distributions with hard assignments as follows:

1. Select initial values for $\mu_k^{(0)}$. Keep $\Sigma = \mathbf{I}$ and $w_k = 1/k$ constant
2. E-step: evaluate new responsibilities $\gamma_{i,k}^{(t)}$ for $1 \leq i \leq N$ and $1 \leq k \leq K$ using current parameters

$$\gamma_{i,k}^{(t)} = \begin{cases} 1 & \text{if } k = \underset{l}{\operatorname{argmin}} \|x_i - \mu_l\|_2^2 \\ 0 & \text{otherwise} \end{cases}$$

3. M-step: evaluate new parameters $\mu_k^{(t+1)}$ for $1 \leq k \leq K$ using current responsibilities

$$\mu_k^{(t+1)} = \frac{\sum_i \gamma_{i,k}^{(t)} \cdot x_i}{\sum_i \gamma_{i,k}^{(t)}}$$

4. Repeat E-step and M-step until the parameters stop changing

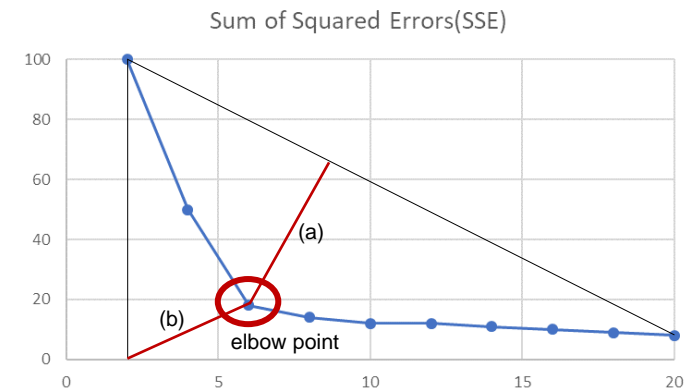
- For both k-means and EM, we need to control the number K of clusters. If the number is too small, the error value is high and the algorithms suffer from underfitting. If we select a large K , we can reduce the error but at risk of overfitting. Let \mathbb{S}_k be the set of data items x that are assigned to cluster C_k . To control K , we determine the **sum of squared errors** SSE over all clusters:

$$SSE(\text{k-means}) = \sum_{k=1}^K \sum_{x \in \mathbb{S}_k} \|x - \mu_k\|_2^2 \quad SSE(\text{EM}) = \sum_{k=1}^K \sum_{x \in \mathbb{S}_k} (x - \mu_k)^T \Sigma_k^{-1} (x - \mu_k)$$

If we plot this SSE as a function of K , we obtain a graph like on the right side below. As we increase the number K , the SSE decreases. However, we cannot simply solve for K that minimizes the SSE function as $K = N$ would have an $SSE = 0$ but clearly overfits the data. Rather, we look for the so-called elbow point as highlighted in the figure where the SSE-functions “abruptly” levels out as is decreasing much slower than before the elbow. We can obtain an optimal K in two ways:

- Vary K from 2 to an upper bound (here 20) and determine the point that lies farthest away from the line between the start and the end of the curve.
- Start with $K = 2$ and determine the distance to the point (2,0). While increasing K observe the distance. Stop if the distance starts growing.

Method b) has the advantage of iterating less over K . For both variants to work, we need to normalize the two dimensions, for instance with a min/max scaling, to obtain a meaningful result.



- **Example: Image Segmentation (Blobworld)**

- Blobworld was a project at the University of Berkeley and published first in 1999. It was using segmentation to divide an image into distinct regions and used descriptors on these regions to retrieve objects embedded in images. The right hand side shows an example of the segmentation

- The original image contains too many edges and corners yielding a large number of potential regions
- A rough Gaussian filter smooths the image and eliminates finer structures
- Color is transformed into the $L^*a^*b^*$ space. For each pixel, Blobworld extract additional texture features describing the polarity (clear direction of edges in a neighborhood), edgeness, and texture contrast. The feature vector consists of the pixel position (x, y) , the 3 color and the 3 texture values at that position.
- Apply the EM algorithm on a Gaussian mixture model over the 8 feature values. This is computed for 2, 3, 4, and 5 clusters.
- To steer the number of clusters, a special objective function based on the Minimum Description Length (MDL) was applied.
- Blobworld hard assigns pixels to a cluster and selects a unique color for each cluster.

(a) original image



(b) smoothed image



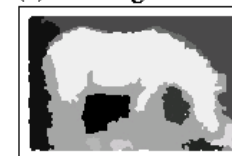
(c) pixel features



(d) EM results: 2, 3, 4, 5 groups



(e) final segmentation

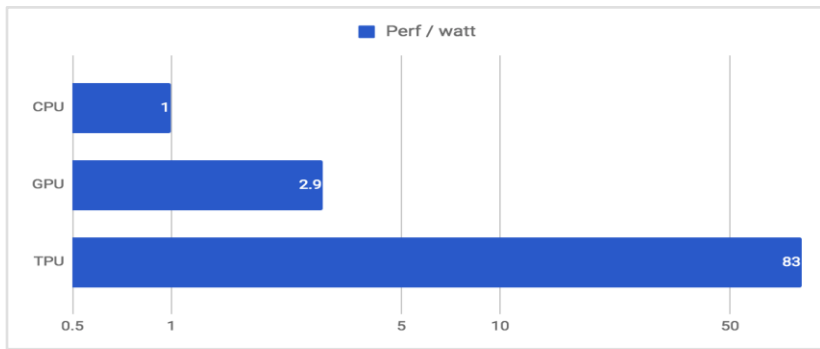


(f) Blobworld



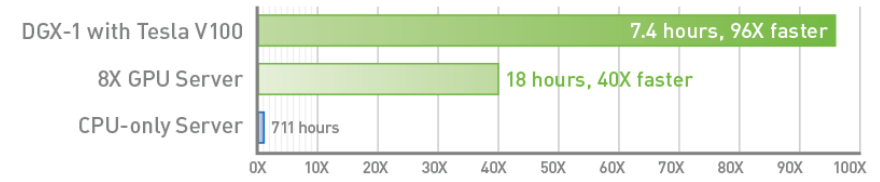
4.9 Simple Neural Network Classifier

- Artificial neural networks are machine learning models that are inspired by how the brain works. Indeed, brain research has frequently led to new approaches like the use of connections between neurons of different layers rather than adjacent ones (multi-layer approach). Neural network, on the other hand, are often employed to model the brain and its learning algorithms.
- The first wave of neural network research started in the late 1950s and was focusing on a single perceptron (in hardware). It was possible to use multiple perceptrons in parallel, but they were only connected to input and output states. The problem of perceptrons was articulated in its famous inability to learn a simple XOR function. Even though it was shown that a two-layer network could indeed encode an XOR function, the limitations were obvious and a first AI winter began.
- The second wave started with research in the 1960s with the introduction of hidden layers. Several researchers were developing similar ideas but the credits usually go to Rumelhart, Hinton, and Williams and their 1986 paper on backpropagation which describes the approach with such clarity that it is still the basis for many descriptions in text books. The area revived quickly and led to convolutional networks, recurrent networks, belief networks with many of the concepts found today in deep learning. However, the field suffered from calculation issues (vanishing and exploding gradients) and the computational limitations in the 1980s and 1990s.
- At the beginning of the 2000s, almost no research was published or cited and funding was very sparse. However, the Canadian government funded a small research team around Hinton that first rebranded the field into “Deep Learning” and then published in 2006 a break-through paper with a fast learning algorithm for deep belief nets. In parallel, compute power has significantly grown. Inspired by the Canadian research team, the field arose again and soon it was found that GPUs were up to 100 times faster than CPUs. This allowed the training of deep networks within hours and days rather than weeks and months. Google started in 2011 its Google Brain research project to connect thousand of CPUs for a network with 1 billion weights. Since then, research has generated an enormous amount of improvements and efficient learning frameworks leading to an overwhelming success story of AI with many applications.



source: <https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu>

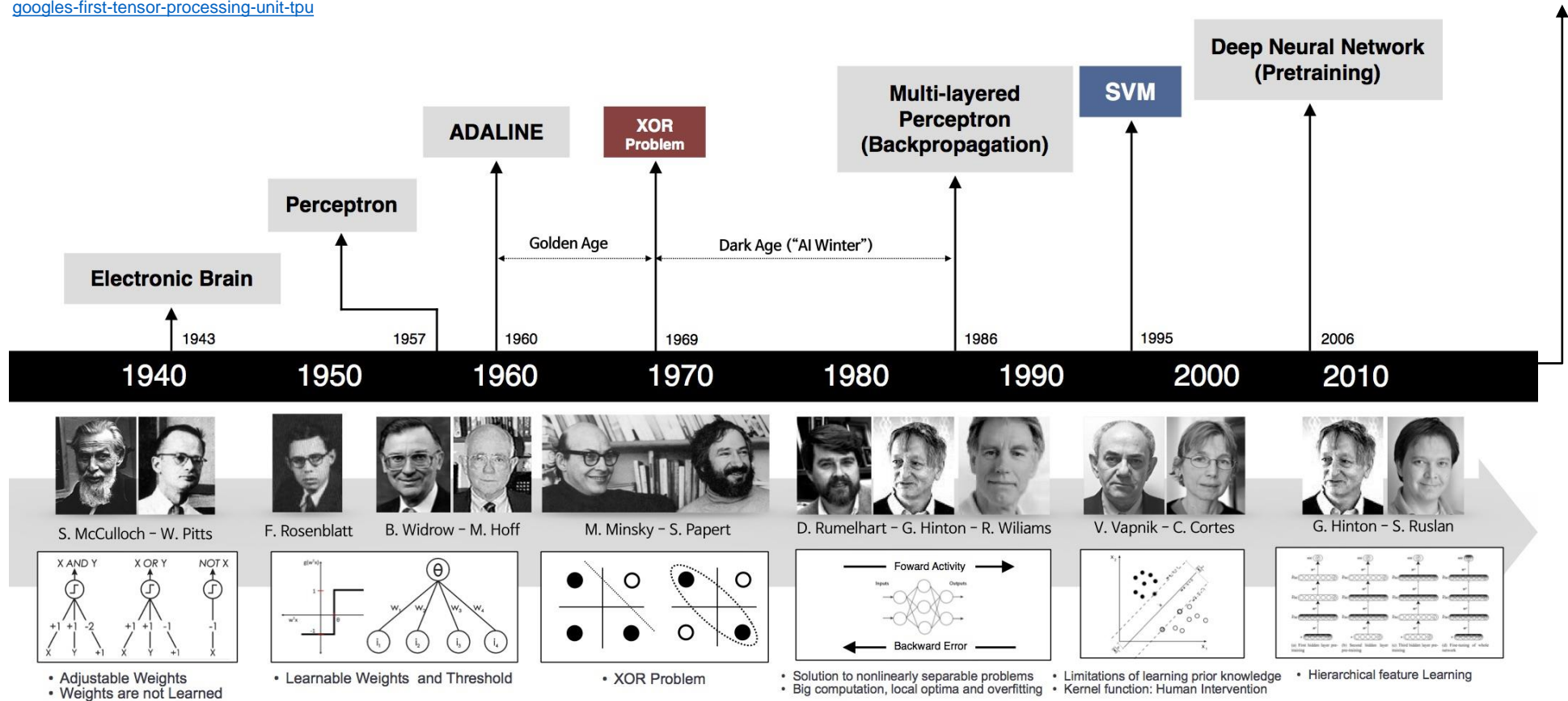
NVIDIA DGX-1 Delivers 96X Faster Training



Relative Performance (Base on Time to Train)

Workload: ResNet50, 90 epochs to solution | CPU Server: Dual Xeon E5-2699 v4, 2.6GHz

source: <https://www.nvidia.com/en-us/data-center/dgx-server/>

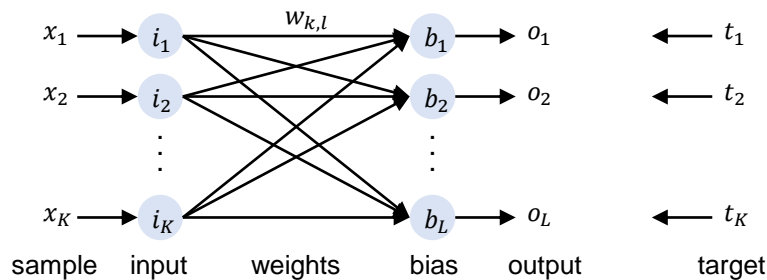


source: [tps://beamandrew.github.io/deeplearning/2017/02/23/deep_learning_101_part1.html](https://beamandrew.github.io/deeplearning/2017/02/23/deep_learning_101_part1.html)

- We first consider the original perceptron idea: in principle, it is a binary classifier mapping a real-valued input vector $\mathbf{x} \in \mathbb{R}^K$ to a binary output value $f(\mathbf{x})$:

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w}^\top \mathbf{x} + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

where $\mathbf{w} \in \mathbb{R}^K$ are the weights and b is the bias. From this definition we derive that the perceptron is splitting the space with a hyperplane given by $\mathbf{w}^\top \mathbf{x} + b$. In a more general setup, L perceptrons with weights \mathbf{w}_l and bias b_l are connected to the K input value i_k and produce L binary output values o_l . We can visualize this general setup as follows:



$$\forall 1 \leq l \leq L: o_l = f\left(\sum_{k=1}^K i_k \cdot w_{k,l} + b_l\right)$$

with the binary step function

$$f(z) = \begin{cases} 1 & z > 0 \\ 0 & \text{otherwise} \end{cases}$$

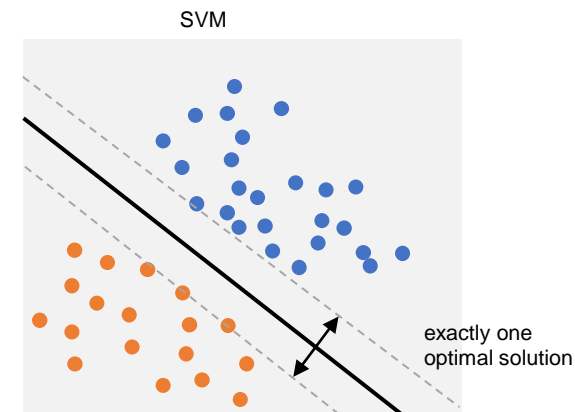
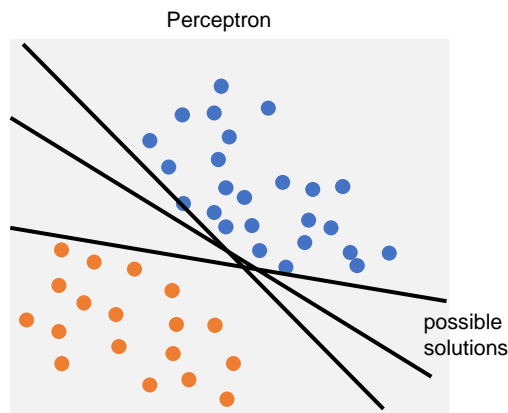
The learning algorithm is then as follows:

(demo: <https://www.cs.utexas.edu/~teammco/misc/perceptron/>)

1. Initialize the weights $w_{k,l}^{(0)}$ and the biases $b_l^{(0)}$ with small random values. Set a learning rate $0 \leq \alpha \leq 1$
2. For each example $\mathbf{x} \in \mathbb{T}$, apply it to the perceptron, i.e., let $\mathbf{i} = \mathbf{x}$
 - Calculate that actual output: $o_l = f\left(\sum_{k=1}^K i_k \cdot w_{k,l} + b_l\right)$
 - Update the weights: $w_{k,l}^{(t+1)} = w_{k,l}^{(t)} + \alpha(t_l - o_l) \cdot i_{k,l}$ (i.e., only adjust if target \neq output)
 - Update the bias: $b_l^{(t+1)} = b_l^{(t)} + \alpha(t_l - o_l)$ (i.e., only adjust if target \neq output)

Convergence is only reached if the data set is linearly separable. Otherwise, the algorithm may fail completely. A number of variants address this later issue.

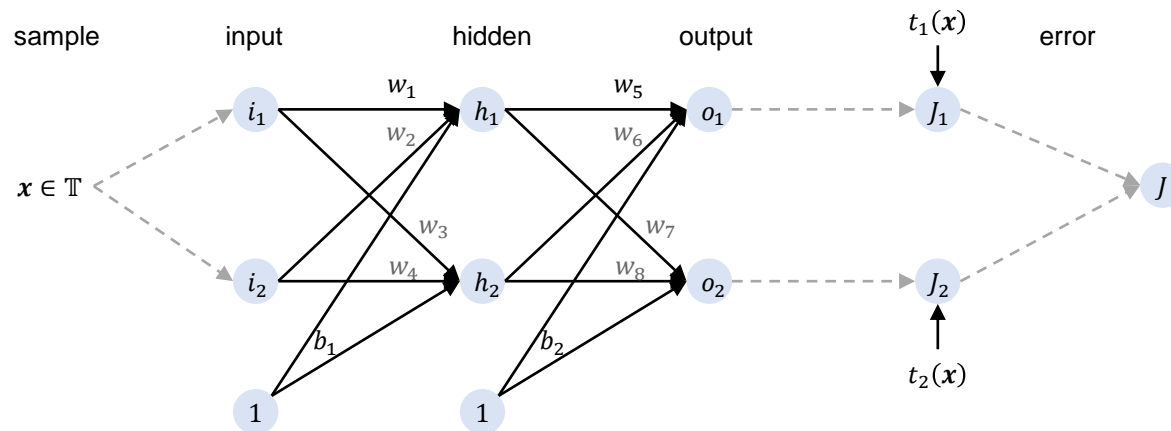
- Intuitively, the perceptron learning algorithm only adjust weights (and bias) if the target differs from the output. If the output is 0 but the target is 1, then weights and bias are incremented, otherwise they are decremented (assuming $x_i \geq 0$). We also note that the algorithm does not aim to optimize any objective function but merely is a heuristic approach to learn the weights. If data is separable, it converges to binary partition of the space with a hyperplane (one of many that partition the space).
- In contrast, the **support vector machine (SVM)** computes an optimal solution for the hyperplane that separates the sets and maximizes the margin (the distance of marginal points to the hyperplane). SVM even works if the data is not separable; it then finds a solution that minimizes the partitioning error. We are not considering here how SVMs are computed.



- In any case, a binary classifier can be used to learn multiclass outputs as well. The “one-vs-all” approach learns a binary classifier for each of the L classes to separate a class C_l from the rest. In other words, we use L perceptrons and the binary target vector \mathbf{t} has $t_l = 1$ and all other components are 0. For prediction, the output with the highest value denotes the “winning” class. Alternatively, the “one-vs-one” strategy uses $L(L - 1)/2$ perceptrons to separate two classes from each other learning the perceptrons individually. For prediction, the output with the highest value indicates the “winning” class.

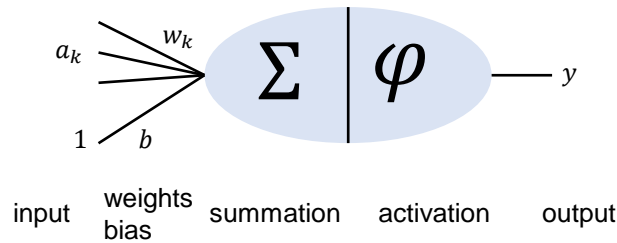
- The linear classification approach of SVM seems rather limiting (like for perceptron). However, SVM has the “kernel trick”: the idea is that data points are mapped to a higher dimensional space that enables better separability of the data by a hyperspace. The mapping to this higher dimensional space is typically non-linear. The “kernel-trick” now means that we do not explicitly compute the mapping to the high-dimensional space, but rather only compute the inner product between data points that is required for the SVM calculations. For instance, the kernel $K(\mathbf{x}, \mathbf{y}) = (1 + \mathbf{x}^\top \mathbf{y})^2$ with $\mathbf{x}, \mathbf{y} \in \mathbb{R}^2$ is an efficient way to compute the inner product of two mapped values $\varphi(\mathbf{x})$ and $\varphi(\mathbf{y})$ in a 6-dimensional space. With a Gaussian kernel $K(\mathbf{x}, \mathbf{y}) = \exp(-\gamma \|\mathbf{x} - \mathbf{y}\|^2)$ we obtain an infinite-dimensional mapping function φ .
- The “kernel trick” is often considered as a human intervention into the machine learning process. SVM classification works very well and is efficient but we need to design an appropriate kernel function for the problem at hand.

- **Multilayer networks** introduce a number of changes to the original perceptron
 - several “hidden” layers between input and output
 - different activation functions to “fire” a neuron, and not necessarily only binary output
 - objective functions to define an optimal state for all network parameters
 - a new algorithm to learn the weights (the so-called **backpropagation**)
- Let us start with a simple two-layer network to understand the fundamentals with a concrete example, and then we generalize the concepts to arbitrary shaped networks.



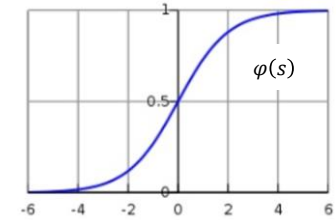
- The network consists of two input neurons i_1, i_2 , two hidden neurons h_1, h_2 and two output neurons o_1, o_2 . We have two (shared) biases, b_1 for the hidden neurons and b_2 for the output neurons. Note that we modeled the bias as a weight from a neuron that always has the state 1. w_1, \dots, w_8 denote the weights on the connections. Even though we have 6 neurons, the connections are only from one layer to the next one and especially, there are no inter-layer connections or cycles. This is an important topological constraint that will simplify our learning algorithm. Finally, we added nodes to capture the training error: J_1 and J_2 measure the error between the first and the second target component $t_1(x)$ and the computed output of the network. J denotes the training error.

- **Feed-Forward:** given a data sample x from the training set \mathbb{T} , the network is computing the state of each neuron using a simple model:



$$s = \sum_k a_k \cdot w_k + b$$

$$y = \varphi(s) = \frac{1}{1 + e^{-s}}$$



We use s to indicate the result of the summation, and we employ the logistic activation function φ also known as soft step. With this, we can determine every state of a neuron, given the input $x \in \mathbb{T}$:

$$i_1 = x_1 \quad \text{and} \quad i_2 = x_2$$

$$h_1 = \varphi(s_{h_1}) = \varphi(w_1 \cdot x_1 + w_2 \cdot x_2 + b_1) \quad \text{and} \quad h_2 = \varphi(s_{h_2}) = \varphi(w_3 \cdot x_1 + w_4 \cdot x_2 + b_1)$$

$$o_1 = \varphi(s_{o_1}) = \varphi(w_5 \cdot h_1 + w_6 \cdot h_2 + b_2) = \varphi(w_5 \cdot \varphi(w_1 \cdot x_1 + w_2 \cdot x_2 + b_1) + w_6 \cdot \varphi(w_3 \cdot x_1 + w_4 \cdot x_2 + b_1) + b_2)$$

$$o_2 = \varphi(s_{o_2}) = \varphi(w_7 \cdot h_1 + w_8 \cdot h_2 + b_2) = \varphi(w_7 \cdot \varphi(w_1 \cdot x_1 + w_2 \cdot x_2 + b_1) + w_8 \cdot \varphi(w_3 \cdot x_1 + w_4 \cdot x_2 + b_1) + b_2)$$

$$\varphi(s) = \frac{1}{1 + e^{-s}}$$

The calculations are straightforward. The term feed-forward denotes that we “feed” the data sample first into the input layer, and then forward the results from one layer to the next one. Each layer can be computed concurrently.

Later on, we will see different activation functions and also different approaches to connectivity and sharing of weights between subsequent layers. The principle model for neurons remain the same for most deep networks. We will also encounter special dropout neurons, that set input elements to zero with a certain probability to prevent overfitting of the network.

- **Error function:** we want to measure how well the network is able to predict the targets for all given data samples in the training set \mathbb{T} . As a starting point, we use the mean square error (MSE):

$$J(\theta) = \frac{1}{|\mathbb{T}|} \sum_{x \in \mathbb{T}} J(x; \theta) = \frac{1}{2 \cdot |\mathbb{T}|} \sum_{x \in \mathbb{T}} \|t(x) - o(x; \theta)\|_2^2$$

where θ denotes the parameters of the network. In our example: $\theta = (w_1, \dots, w_8, b_1, b_2)$. Learning a network means finding parameters θ^* that minimizes the error function:

$$\theta^* = \underset{\theta}{\operatorname{argmin}} J(\theta) = \frac{1}{2 \cdot |\mathbb{T}|} \sum_{x \in \mathbb{T}} \|t(x) - o(x; \theta)\|_2^2$$

- Due to the size of networks and the number of data items, it is generally not feasible to solve the equation in closed form. Instead, we use the gradient descent method to find a (local) optimum through an iterative approach. Let $\nabla J(\theta)$ be the gradient of $J(\theta)$ for the parameters θ of the network. The **gradient descent** method defines the learning strategy for the network:

1. Choose an initial random vector for $\theta^{(0)}$ and a learning rate $0 \leq \eta \leq 1$
2. Repeat until $\|\theta^{(t+1)} - \theta^{(t)}\|_2^2 \leq \varepsilon$ or $t > t_{max}$
 - Compute gradient: $\Delta^{(t)} = \eta \cdot \nabla J(\theta^{(t)})$
 - Adjust parameters: $\theta^{(t+1)} = \theta^{(t)} - \Delta^{(t)}$

- Gradient descent is relatively slow close to the minimum and often “zigzags” for poorly conditioned convex functions. In addition, for large-scale data sets and networks, gradient descent requires enormous computational and storage requirements to determine the gradient (which we can derive in closed form for the network as we will see later).

- Instead of gradient descent, neural network algorithms use the **stochastic gradient descent (SGD)** often in combination with a momentum method to prevent the afore mentioned zigzag issue. SGD approximates the true gradient of $J(\theta)$ with a single data sample (instead of over all data samples). As we will see with backpropagation, this allows us to quickly update the weights with minimal storage overhead. SGD still suffers from slow convergence especially towards the end of the iterations. Momentum is one method to accelerate the descent. We keep the gradient of the past iteration and re-apply some fraction γ of it in the descent:

1. Choose an initial random vector for $\theta^{(0)}$, a learning rate $0 \leq \eta \leq 1$, and a momentum $0 \leq \gamma \leq 1$.
2. Repeat until $\|\theta^{(t+1)} - \theta^{(t)}\|_2^2 \leq \varepsilon$ or $t > t_{max}$
 - Randomly shuffle the training set \mathbb{T}
 - $\theta^{(t+1)} = \theta^{(t)}$
 - For each $x \in \mathbb{T}$
 - Compute gradient: $\Delta = \gamma \cdot \Delta + \eta \cdot \nabla J(x; \theta^{(t+1)})$
 - Adjust parameters: $\theta^{(t+1)} = \theta^{(t+1)} - \Delta$
 - Increase γ

The momentum γ defines how long a previous gradient is still used. Generally, we start with $\gamma = 0.5$ and then increase it after the initial learning stabilizes to $\gamma = 0.9$ or even higher.

- The above algorithm defines the overall learning strategy. Each batch (step 2) runs against the entire training set and for each data samples, the weights and biases in the network are adjusted for each data sample. What remains to do is to compute the gradient $\nabla J(x; \theta)$ for the current data sample and the current set of parameters of the network.

$$J(x; \theta) = \frac{1}{2} \|t(x) - o(x; \theta)\|_2^2$$

$$\nabla J(x; \theta) = ?$$

- **Gradient computation:** before we consider the backpropagation algorithm, let us re-consider our example network from the beginning with two input nodes, two hidden nodes, and two output nodes. For the stochastic gradient descent, we need to compute the gradient. Note that in our example, we have $\theta = (w_1, \dots, w_8, b_1, b_2)$. The gradient is then given as the partial derivatives over $J(x; \theta)$:

$$\nabla J(x; \theta) = \left(\frac{\partial J}{\partial w_1}, \dots, \frac{\partial J}{\partial w_8}, \frac{\partial J}{\partial b_1}, \frac{\partial J}{\partial b_2} \right) \quad J(x; \theta) = J_1(x; \theta) + J_2(x; \theta) = \frac{1}{2} \cdot (t_1 - o_1)^2 + \frac{1}{2} \cdot (t_2 - o_2)^2$$

with given targets t_1 and t_2 for data sample x , and o_1 and o_2 as given previously as a function of x and the weights w_1, \dots, w_8 and the biases b_1 and b_2 .

- Let us start simple: consider w_5 . It only occurs in o_1 but not in o_2 . Thus the partial derivative is:

$$o_1 = \varphi(s_{o_1}) = \varphi(w_5 \cdot h_1 + w_6 \cdot h_2 + b_2) \quad o_2 = \varphi(s_{o_2}) = \varphi(w_7 \cdot h_1 + w_8 \cdot h_2 + b_2)$$

$$\frac{\partial J}{\partial w_5} = \frac{\partial}{\partial w_5} \left(\frac{1}{2} \cdot (t_1 - o_1)^2 + \frac{1}{2} \cdot (t_2 - o_2)^2 \right) = \frac{\partial}{\partial w_5} \left(\frac{1}{2} \cdot (t_1 - o_1)^2 \right) = (t_1 - o_1) \cdot \frac{\partial o_1}{\partial w_5}$$

$$\frac{\partial o_1}{\partial w_5} = \frac{\partial}{\partial w_5} (\varphi(s_{o_1})) = \varphi(s_{o_1}) \cdot (1 - \varphi(s_{o_1})) \cdot \frac{\partial s_{o_1}}{\partial w_5} = o_1 \cdot (1 - o_1) \cdot \frac{\partial s_{o_1}}{\partial w_5}$$

$$\frac{\partial s_{o_1}}{\partial w_5} = \frac{\partial}{\partial w_5} (w_5 \cdot h_1 + w_6 \cdot h_2 + b_2) = h_1$$

all together:

$$\frac{\partial J}{\partial w_5} = (t_1 - o_1) \cdot o_1(1 - o_1) \cdot h_1$$

$$\varphi(s) = \frac{1}{1 + e^{-s}} \\ \varphi' = \varphi \cdot (1 - \varphi)$$

- Similarly, we obtain the other partial derivatives $\frac{\partial J}{\partial w_6}$, $\frac{\partial J}{\partial w_7}$, $\frac{\partial J}{\partial w_8}$, and $\frac{\partial J}{\partial b_2}$. Altogether, we have:

$$\begin{aligned}\frac{\partial J}{\partial w_5} &= (t_1 - o_1) \cdot o_1(1 - o_1) \cdot h_1 & \frac{\partial J}{\partial w_6} &= (t_1 - o_1) \cdot o_1(1 - o_1) \cdot h_2 \\ \frac{\partial J}{\partial w_7} &= (t_2 - o_2) \cdot o_2(1 - o_2) \cdot h_1 & \frac{\partial J}{\partial w_8} &= (t_2 - o_2) \cdot o_2(1 - o_2) \cdot h_2 \\ \frac{\partial J}{\partial b_2} &= (t_1 - o_1) \cdot o_1(1 - o_1) + (t_2 - o_2) \cdot o_2(1 - o_2)\end{aligned}$$

We already note the recurring patterns in the calculations: the derivatives on the error function are multiplied by the derivative on the activation function and are multiplied by the derivative on the summation. For the gradients, we require the results (=states) from the feed-forward step and can then efficiently compute the gradients (see backpropagation).

- Now to the remaining partial derivatives (see next page how to derive for w_1):

$$\begin{aligned}\frac{\partial J}{\partial w_1} &= h_1 \cdot (1 - h_1) \cdot x_1 \cdot ((t_1 - o_1) \cdot o_1 \cdot (1 - o_1) \cdot w_5 + (t_2 - o_2) \cdot o_2 \cdot (1 - o_2) \cdot w_7) \\ \frac{\partial J}{\partial w_2} &= h_1 \cdot (1 - h_1) \cdot x_2 \cdot ((t_1 - o_1) \cdot o_1 \cdot (1 - o_1) \cdot w_5 + (t_2 - o_2) \cdot o_2 \cdot (1 - o_2) \cdot w_7) \\ \frac{\partial J}{\partial w_3} &= h_2 \cdot (1 - h_2) \cdot x_1 \cdot ((t_1 - o_1) \cdot o_1 \cdot (1 - o_1) \cdot w_6 + (t_2 - o_2) \cdot o_2 \cdot (1 - o_2) \cdot w_8) \\ \frac{\partial J}{\partial w_4} &= h_2 \cdot (1 - h_2) \cdot x_2 \cdot ((t_1 - o_1) \cdot o_1 \cdot (1 - o_1) \cdot w_6 + (t_2 - o_2) \cdot o_2 \cdot (1 - o_2) \cdot w_8) \\ \frac{\partial J}{\partial b_1} &= h_1 \cdot (1 - h_1) \cdot ((t_1 - o_1) \cdot o_1 \cdot (1 - o_1) \cdot w_5 + (t_2 - o_2) \cdot o_2 \cdot (1 - o_2) \cdot w_7) + \\ &\quad h_2 \cdot (1 - h_2) \cdot ((t_1 - o_1) \cdot o_1 \cdot (1 - o_1) \cdot w_6 + (t_2 - o_2) \cdot o_2 \cdot (1 - o_2) \cdot w_8)\end{aligned}$$

- Let us now consider w_1 : we note that w_1 only occurs in h_1 which in turn is part of both o_1 and o_2 .

$$o_1 = \varphi(s_{o_1}) = \varphi(w_5 \cdot h_1 + w_6 \cdot h_2 + b_2)$$

$$o_2 = \varphi(s_{o_2}) = \varphi(w_7 \cdot h_1 + w_8 \cdot h_2 + b_2)$$

$$h_1 = \varphi(s_{h_1}) = \varphi(w_1 \cdot x_1 + w_2 \cdot x_2 + b_1)$$

$$h_2 = \varphi(s_{h_2}) = \varphi(w_3 \cdot x_1 + w_4 \cdot x_2 + b_1)$$

$$\frac{\partial J}{\partial w_1} = \frac{\partial}{\partial w_1} \left(\frac{1}{2} \cdot (t_1 - o_1)^2 + \frac{1}{2} \cdot (t_2 - o_2)^2 \right) = (t_1 - o_1) \cdot \frac{\partial o_1}{\partial w_1} + (t_2 - o_2) \cdot \frac{\partial o_2}{\partial w_1}$$

$$\frac{\partial o_1}{\partial w_1} = \frac{\partial}{\partial w_1} (\varphi(s_{o_1})) = \varphi(s_{o_1}) \cdot (1 - \varphi(s_{o_1})) \cdot \frac{\partial s_{o_1}}{\partial w_1} = o_1 \cdot (1 - o_1) \cdot \frac{\partial s_{o_1}}{\partial w_1}$$

$$\frac{\partial o_2}{\partial w_1} = o_2 \cdot (1 - o_2) \cdot \frac{\partial s_{o_2}}{\partial w_1}$$

$$\frac{\partial s_{o_1}}{\partial w_1} = \frac{\partial}{\partial w_1} (w_5 \cdot h_1 + w_6 \cdot h_2 + b_2) = w_5 \cdot \frac{\partial h_1}{\partial w_1}$$

$$\frac{\partial s_{o_2}}{\partial w_1} = w_7 \cdot \frac{\partial h_1}{\partial w_1}$$

$$\frac{\partial h_1}{\partial w_1} = \frac{\partial}{\partial w_1} (\varphi(s_{h_1})) = \varphi(s_{h_1}) \cdot (1 - \varphi(s_{h_1})) \cdot \frac{\partial s_{h_1}}{\partial w_1} = h_1 \cdot (1 - h_1) \cdot \frac{\partial s_{h_1}}{\partial w_1}$$

$$\frac{\partial s_{h_1}}{\partial w_1} = \frac{\partial}{\partial w_1} (w_1 \cdot x_1 + w_2 \cdot x_2 + b_1) = x_1$$

all together:

$$\begin{aligned} \frac{\partial J}{\partial w_1} = & (t_1 - o_1) \cdot o_1 \cdot (1 - o_1) \cdot w_5 \cdot h_1 \cdot (1 - h_1) \cdot x_1 + \\ & (t_2 - o_2) \cdot o_2 \cdot (1 - o_2) \cdot w_7 \cdot h_1 \cdot (1 - h_1) \cdot x_1 \end{aligned}$$

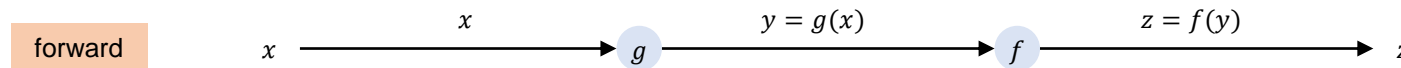
$$\begin{aligned} \varphi(s) &= \frac{1}{1 + e^{-s}} \\ \varphi' &= \varphi \cdot (1 - \varphi) \end{aligned}$$

- Evidentially, it is possible to compute all partial derivatives for the gradient, but it seems tedious work to do so (and error prone). Can we do it simpler? Yes, we can. Backpropagation is an astonishingly simple scheme that computes the gradient starting at the error node and working back towards the input nodes. It does not provide us with the closed forms of the derivatives, but it computes the gradient avoiding multiple computations of the same sub-expressions.
 - Let us look again at the chain rule from calculus:

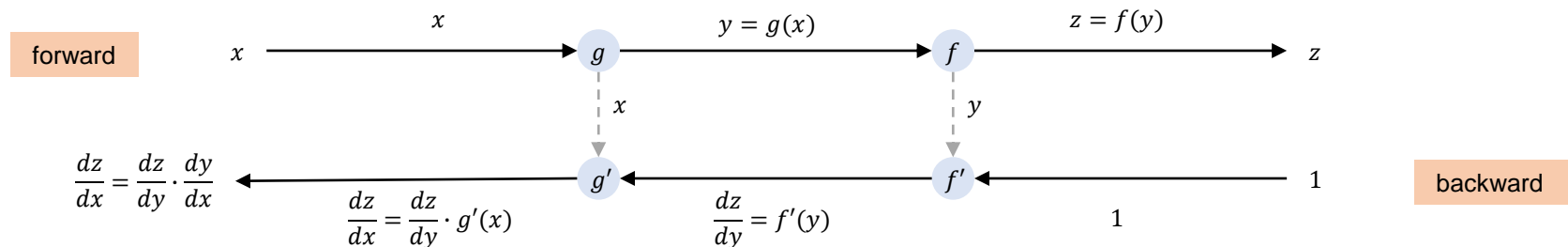
$$F(x) = f \circ g = f(g(x)) \qquad F'(x) = f'(g(x)) \cdot g'(x)$$

or in Leibniz notation with $z = f(y)$ and $y = g(x)$: $\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} = f'(y) \cdot g'(x)$

In graphical notation, we obtain the forward path to compute the composite function:



Now to compute the derivative $\frac{dz}{dx}$ for x we move backwards. We first compute $f'(y)$ and then multiply it with $g'(x)$. To this end, we need to keep track of intermediate results and use them on the back path to calculate the derivative:



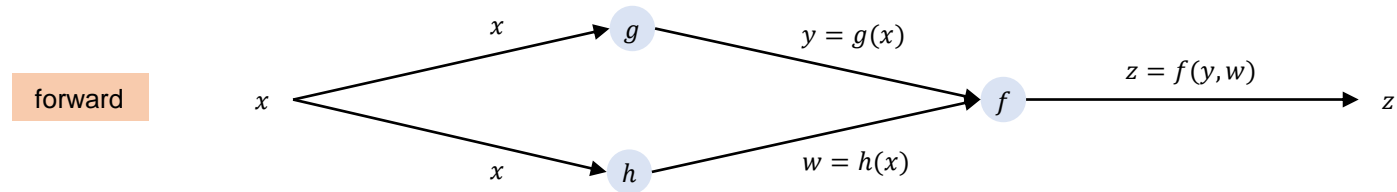
- Similarly, we can look at multivariable chain rules

$$F(x) = f(g(x), h(x)) \quad F'(x) = f'(g(x), h(x)) \cdot g'(x) + f'(g(x), h(x)) \cdot h'(x)$$

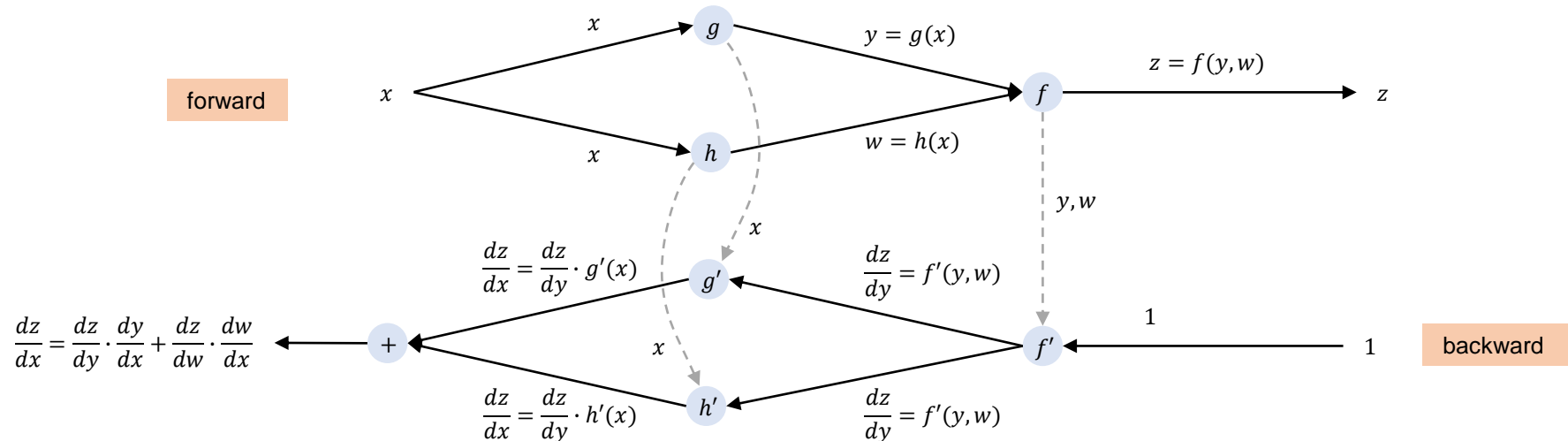
or in Leibniz notation with $z = f(y, w)$, $y = g(x)$ and $w = h(x)$

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} + \frac{dz}{dw} \cdot \frac{dw}{dx} = f'(y, w) \cdot g'(x) + f'(y, w) \cdot h'(x)$$

In graphical notation, we obtain the forward path to compute the function:

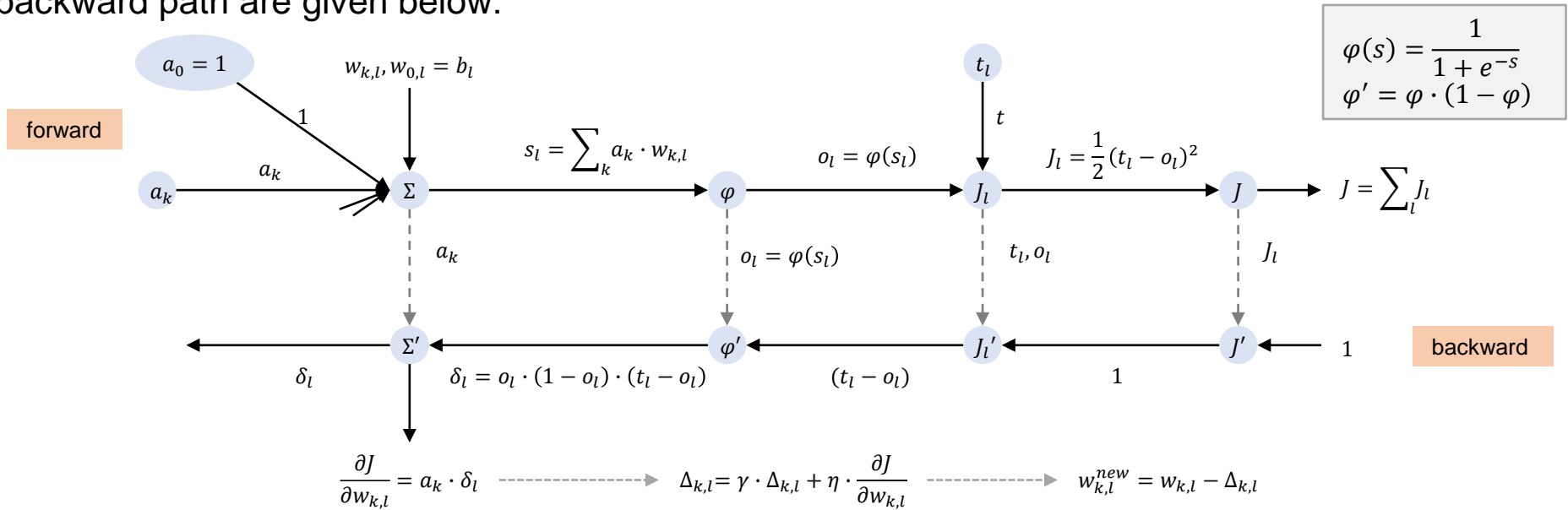


Now to compute the derivative $\frac{dz}{dx}$ for x we move backwards similarly as before:

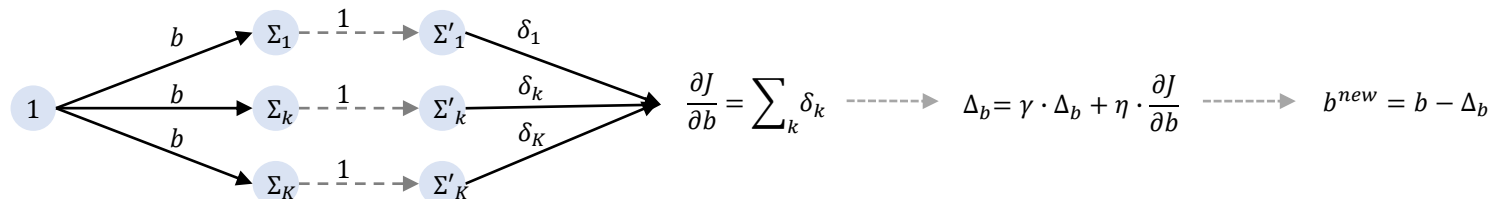


Additional information — not part of the exams

- Let us apply the chain rule to our neural network. Let start with the output neurons. To simplify the structure, we introduce a node a_0 which always has the state 1, and the weight $w_0 = b$ which represents the bias. All formulas become a bit simpler. The visualization for the forward and backward path are given below:

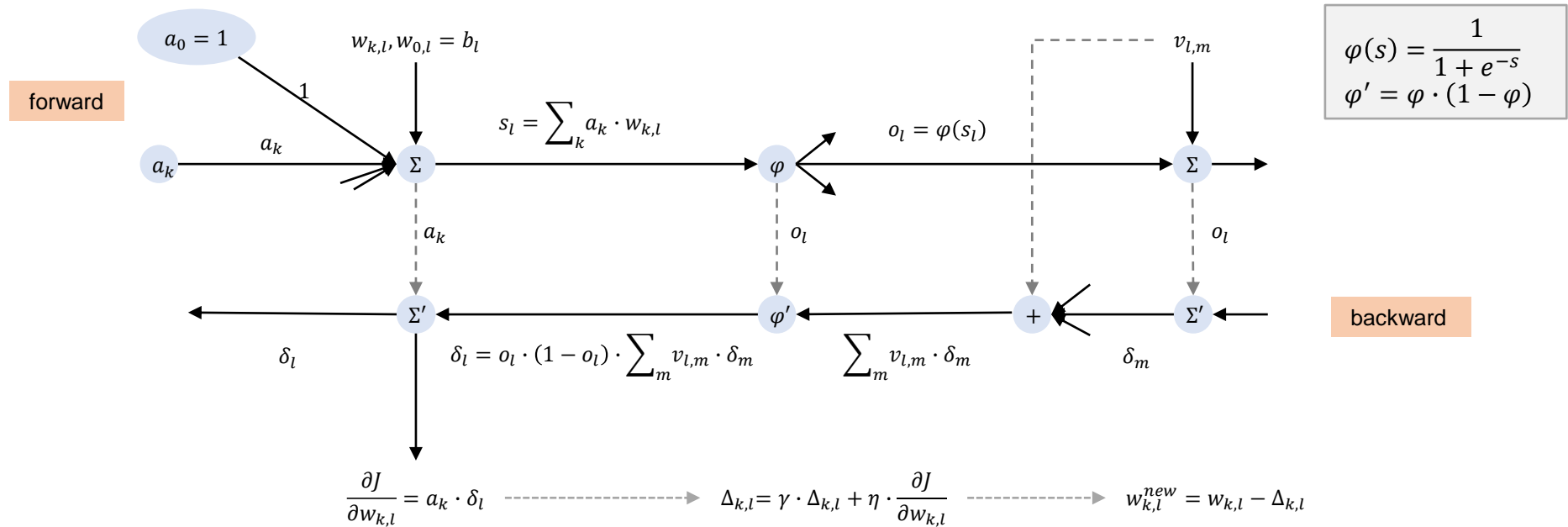


- Every layer outputs the δ -values that are propagated back to the inputs and are used to adjust the parameters in every layer. Above, we used a separate bias b_l for each node. If we would share the bias across the layer like in the example, we need to simply sum up the deltas over the nodes using the same bias, i.e.:



Additional information — not part of the exams

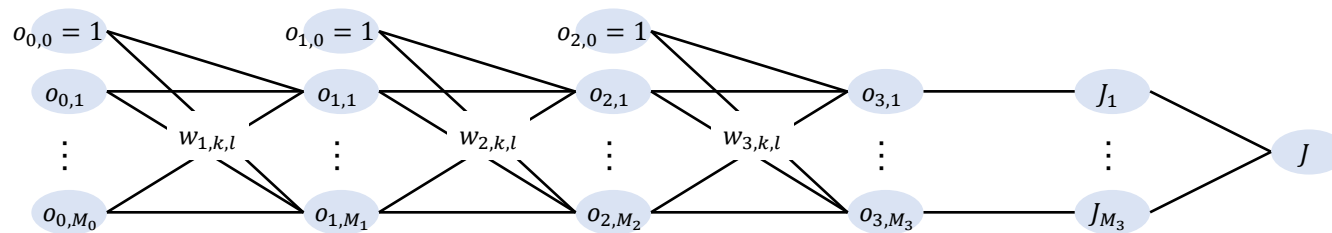
- Hidden layers are calculated similarly, however, there are L incoming edges from the subsequent layer during backpropagation. The visualization for the forward and backward path are as follows:



- Let us sum up the backpropagation algorithm: during the stochastic gradient descent, we search for the optimal parameters (weights, biases, etc.) of the network. To compute the gradient for these parameters with respect to an error function J , we first use the network in forward mode to predict the output with the current set of parameters. At the same time, we keep track of intermediate values that are required on the backward path. We then compute the error with regard to a single sample and propagate the partial derivatives backwards to the previous layers. At each layer, we compute the Δ -values for the weights to obtain new estimates for them. Note that the old weights are still required for the preceding layer to compute its partial derivative (see figure above, the (+)-node requires weights $v_{l,m}$ from the subsequent layer).

- **Generic implementation of multilayer networks:** let us model a dense multilayer network. We assume N layers L_i and we denote L_0 to be the input layer and L_N to be the output layer. Each layer has M_i neurons with states $o_{i,k}$ with $0 \leq i \leq N$ and $0 \leq k \leq M_i$ whereby $o_{i,0} = 1$ (used for the bias). Further we use weights $w_{i,k,l}$ with $1 \leq i \leq N$, $0 \leq k \leq M_i$ and $1 \leq l \leq M_{i-1}$ to connect the l -th node of Layer L_{i-1} with the k -th node of Layer L_i . In addition, we keep track of the increments $\Delta_{i,k,l}$ for the computation of the gradients $\frac{\partial J}{\partial w_{i,k,l}}$.

- Example with 3 layers:



- Feed Forward is then given as:

1. Initialize $o_{0,k} = x_k$ from the current data sample $x \in \mathbb{T} \subset \mathbb{R}^{M_0}$ with target $t \in \mathbb{R}^{M_N}$
2. For each layer L_i with i iterating from 1 to N :
 - Compute $o_{i,k} = \varphi(\sum_l w_{i,k,l} \cdot o_{i-1,l})$ with a selected activation function φ for all $1 \leq k \leq M_i$
3. Compute $J_k = E_k(o_{N,k}; t_k)$ with a selected error function E for all $1 \leq k \leq M_N$
4. Compute training error $J(x; \theta) = \sum_k J_k = E(o_{N,k}; t_k)$ for current sample

So far we have used the logistic activation function $\varphi(s) = \frac{1}{1+e^{-z}}$ and the mean square error (MSE) with $J(\theta) = \frac{1}{2 \cdot |\mathbb{T}|} \sum_{x \in \mathbb{T}} \|t(x) - o(x; \theta)\|_2^2$ such that $E_k(o_{N,k}; t_k) = \frac{1}{2} (t_k - o_{N,k})^2$. We will see further activation functions and error (or loss) functions in the deep learning section.

– Backpropagation is finally (e.g., with logistic activation function and mean square error):

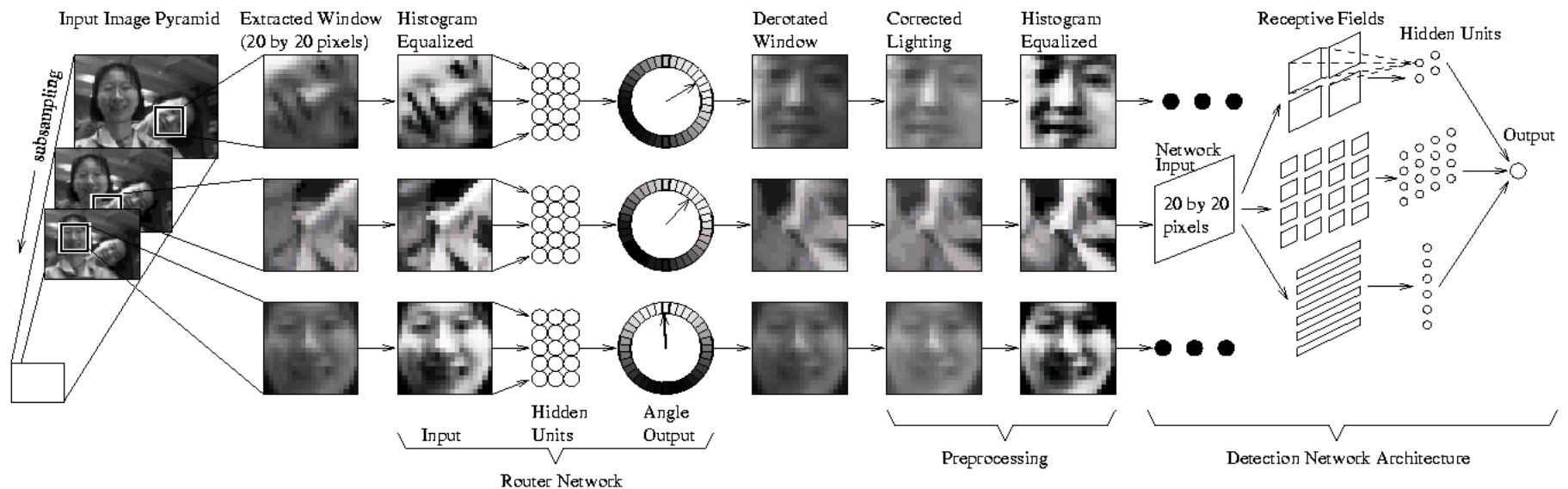
1. Given target t and assume output o_N from feed forward step; assume learning rate η and momentum γ
2. Initialize $\Delta_{i,k,l} = 0$
3. Compute $\delta_{N,k} = \varphi'(o_{N,k}) \cdot E'_k(o_{N,k}; t_k) = o_{N,k} \cdot (1 - o_{N,k}) \cdot (t_k - o_{N,k})$ for all $1 \leq k \leq M_N$
4. For each layer L_i with i iterating from $N - 1$ down to 1:
 - Compute $\delta_{i,k} = \varphi'(o_{i,k}) \cdot \sum_l w_{i+1,l,k} \cdot \delta_{i+1,l}$ for all $1 \leq k \leq M_i$
 - Compute $\Delta_{i,k,l} = \gamma \cdot \Delta_{i,k,l} + \eta \cdot o_{i-1,l} \cdot \delta_{i,k}$ for all $1 \leq k \leq M_i$
5. Update weights $w_{i,k,l} = w_{i,k,l} - \Delta_{i,k,l}$

Note: it is tempting to update the weights in the inner loop (step 4). However, we need the old weights in the preceding layer (next iteration in step 4) to compute $\delta_{i,k}$.

- While multilayer networks are still used in later layers in deep learning scenarios, the original approaches in 1980s and 1990s suffered from a number of issues (we will discuss them in the deep learning section). Essentially, the main issues involved numerical problems while computing the gradients (vanishing and exploding values) and the vast compute power necessary to learn moderate to large network. The smaller networks, on the other hand, did not work too well on typical classification scheme, and with SVM and kernel functions superior alternatives emerged.

- **Example: Face Detection**

- Rowley, Baluja, Kanade [1998], Carnegie Mellon University, defined an elaborated algorithm for detecting faces at any scale and direction. To keep the neural network small, their approach was to first learn only normalized faces, and to then apply an exhaustive search for faces on images. The detection network is based on a 20x20 input network (preprocessed image window). In a first layer, 3 types of receptive fields are created: a) four 10x10 areas, b) 16 5x5 areas, and c) six overlapping 20x5 areas. Each area is fully connected to a hidden unit which is fully connected to an output. An output of 1 denotes a face, and an output of -1 denotes no face.
- A second network (router network) was trained to estimate the direction of a face within a window. The 20x20 input network (preprocessed image window) is fully connected to hidden units which in turn are fully connected to 36 output values representing an angle of $i \cdot 36^\circ$. The angle can be used in the predication phase to normalize the face before application of the detection network.



- Once trained, we can find faces in an image as follows: first, we build a pyramid of images by subsampling to smaller and smaller sizes. This allows us to find faces of different sizes. Then, a 20x20 windows is sliding across the image and for each location, the network tests whether the window contains a face. Due to the usage of normalized faces, the algorithm can return the location and direction of faces as well as estimating the position of the eyes.



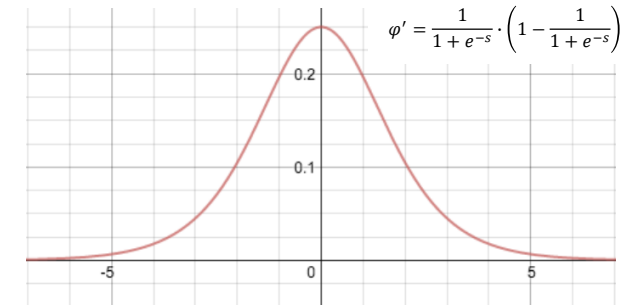
4.10 Deep Learning

- The second wave of neural network research died very quickly after discovering more structural issues with how the learning algorithm works. Even though it was proven that neural networks can learn any function, that theory often would not materialize in practice. Especially, it was observed that adding additional hidden layers does not lead to better results, and bigger networks were becoming increasingly instable to operate. The famous notion of **vanishing and exploding gradients** and the competition of support vector machine (SVM) with elaborated kernels drove a whole research field into a dead end. Only the Canadian government continued to fund neural network research: Geoff Hinton and team published in 2006 a paper on deep belief network where they showed how they could learn a network layer wise overcoming the issues of early backpropagation learning. In parallel, the massive amount of labeled data sets (a prerequisite to start learning) and the massive parallelism of GPUs greatly accelerated the success of what is now simply called deep learning (although the concepts are much older).
- Let us first consider the vanishing gradient problem. In the network of the previous section, we had an input layer, a hidden layer, and an output layer and were optimizing the network's parameters by minimizing a quadratic cost function. The backpropagation algorithm computes gradients and would update a weight on the first layer with:

$$\frac{\partial J}{\partial w_1} = (t_1 - o_1) \cdot o_1 \cdot (1 - o_1) \cdot w_5 \cdot h_1 \cdot (1 - h_1) \cdot x_1 + (t_2 - o_2) \cdot o_2 \cdot (1 - o_2) \cdot w_7 \cdot h_1 \cdot (1 - h_1) \cdot x_1$$

The gradient is the sum of two multiplications, each with factors of the form $x \cdot (1 - x)$ due to the usage of the sigmoid activation function. Note that x stands for the outcome of a neuron after the activation function, hence $x = \varphi(s) = \frac{1}{1+e^{-s}}$. In addition, the multiplications include the weights of the last layer. If we add more hidden layers to the network, more factors of the form $x \cdot (1 - x)$ and more weights of later layers appear in the gradients of weights and bias of the first layer.

- The derivative of the sigmoid function $\varphi(s) = \frac{1}{1+e^{-s}}$ is plotted on the right hand side. We note that the maximum value is $\frac{1}{4}$ and that values quickly drop on both sides. If we initialize weights between 0 and 1, the gradient computation turns into a series of multiplications of small values yielding very small updates weights and biases even if they are significantly wrong. This requires a huge number of iterations to move weights and biases towards their optimal values, hence, learning is very slow and expensive.



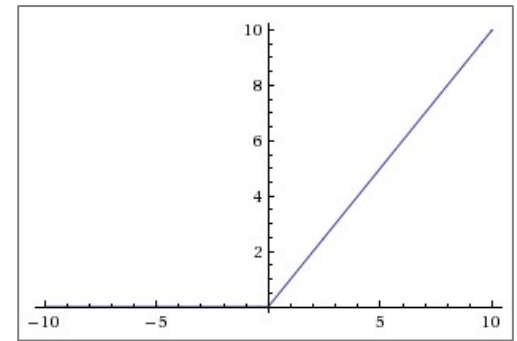
$$(t_1 - o_1) \cdot \underbrace{o_1 \cdot (1 - o_1)}_{\leq 1/4} \cdot \underbrace{w_5}_{\leq 1} \cdot \underbrace{h_1 \cdot (1 - h_1)}_{\leq 1/4} \cdot x_1 \leq 1/16$$

As a consequence, gradients are reduced to a fourth for each layer in the backpropagation making it very slow to train networks with lots of layers (GoogLeNet used ~20 layers).

- On the other hand, if we scale the weights and input values beyond the typical $[-1,1]$ range, the gradients will explode as we now multiply several numbers larger than 1. With only a few layers, gradients become exponentially larger as we propagate back, and with that the weights and biases grow in absolute values, resulting in potentially even larger gradients in the next iteration. Several attempts for deeper networks failed due to instable gradient computations.
- Deep learning addressed these issues with backpropagation friendly activation functions (ReLU), improved architecture (convolution, pooling, inception modules, residual networks), and improved regularization techniques (dropout, ReLU, L1, L2). We consider some of these concepts subsequently.

- The **rectified linear unit (ReLU)** is a simple activation function replacing the sigmoid function used previously. There are now many alternative activation functions, but the ReLU marked an important step towards more stable gradient computations. It is defined as

$$\varphi(s) = \max(0, s)$$



The function is plotted on the right hand side. What is so special about this function? First, its is closer to the way biological neurons works while the sigmoid function (and its counterpart the hyperbolic tangent) were inspired by probability theory. Second, its gradient is either 0 or 1:

$$\varphi'(s) = \begin{cases} 0, & s < 0 \\ 1, & s \geq 0 \end{cases}$$

Hence, the gradients of the activation function do not accelerate the vanishing and exploding effects as described before. ReLU have become the standard activation function for deep learning despite some of the challenges that come with them:

- The output is no longer in the range $[0,1]$. If we train classifiers, how can we map the output of the last layer to class labels? The softmax function can be used to convert output values to class probabilities. It is often used together with the cross-entropy loss function to simplify gradient calculations as follows. Let o_k be the k -th output value, and y_k be the target label. Then:

$$p_k = \frac{e^{o_k}}{\sum_j e^{o_j}}$$

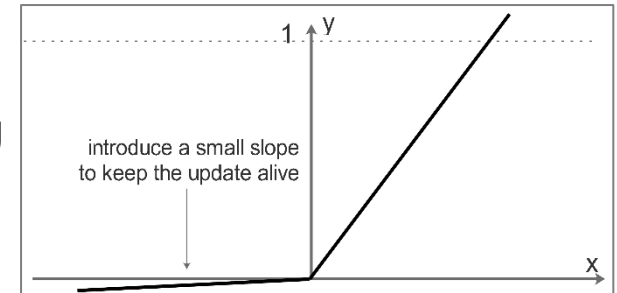
$$J(\theta) = - \sum_k y_k \cdot \log p_k$$

J is defined as the cross-entropy loss function. θ contains all parameters of the network, i.e., weights and biases.

$$\frac{\partial J}{\partial o_k} = p_k - y_k$$

that is simple!

- The derivative of the ReLU can become 0 which means that back propagation stops at this unit and predecessors are not adjusted. While some see this as a regularization of the network by thinning out the connections (much like neurons in the brain are also not fully connected), others are concerned that an initial selection of weights and biases may randomly close paths and the network can only slowly recover from that (if at all). Instead, a common extension is the **leaky ReLU** which is defined as (including its derivative):



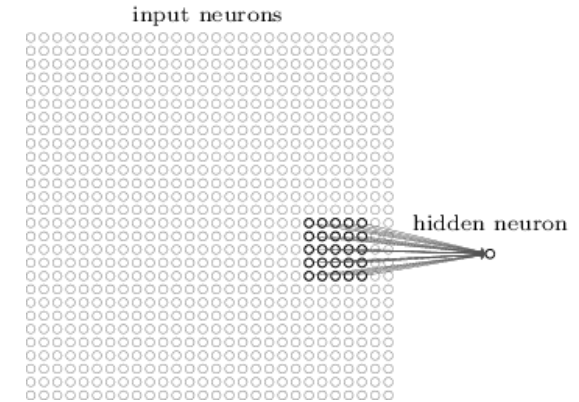
$$\varphi(s) = \begin{cases} 0.01 \cdot s, & s < 0 \\ s, & s \geq 0 \end{cases} \quad \varphi'(s) = \begin{cases} 0.01, & s < 0 \\ 1, & s \geq 0 \end{cases}$$

The advantage is that the derivative is never becoming 0; it is small for negative values allowing a network to recover a closed path

- To overcome the vanishing and exploding gradient, deep learning improved the architecture of the network: instead of fully connected, cascading layers, deep networks uses convolution, pooling, inception, residuals, and regularizations to structure the network. Convolution, for instance, uses a few weights and biases that feed into several thousands output neurons. Hence, during backpropagation, even though the gradients may have become small, thousands of updates are summed up in one iteration. Regularizations, as another example, reduces the number of active connections. Similar to convolutions, this reduces the number of (active) parameters in the network making it more efficient to train and faster to learn. We look at these individual measure first in isolation and then put all together for a truly deep learning network.

- **Convolution**

- So far, we considered layers that were fully connected with the previous layer. Each connection had its own weight, and neurons had either their own bias or a shared bias.
- In contrast, the visual perception of nature works with receptive fields that extract features from a spatial neighborhood. The fields work the same across the entire visual range. In the traditional learning, hence, images were pre-processed using different algorithms (Gaussian, Sobel, HOG). However, that also limited the ways a network can learn.
- Deep learning introduced a new layer, the convolutional layer. As depicted above, it connects only a small spatial neighborhood (here 5x5 input neurons) to a hidden neuron. This occurs for all locations in the matrix, creating an identically sized hidden layer (using padding at the boundaries). The output of the neuron is given as:



$$o_{i,j}(\mathbf{x}) = \varphi \left(b + \sum_{k,l} w_{k,l} \cdot x_{i+k,j+l} \right)$$

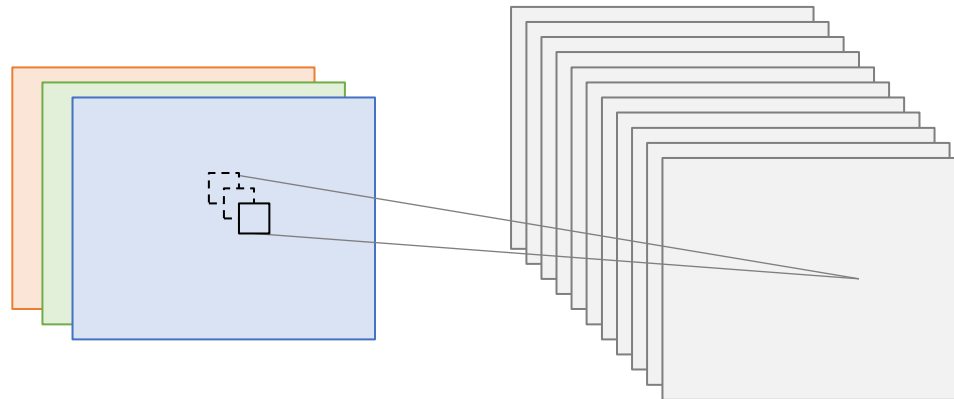
An interesting aspect is that the weights $w_{k,l}$ and the bias b are shared across the neurons of the new layer. In fact, the above formula correspond to the convolution approach we have seen in the previous chapter (hence the name). Only, here we task the network to learn the best convolution for the task at hand.

- In addition, we can define an arbitrary number of such filters within a single convolution layer. The output at the hidden neuron is then not only a single value, but a N -dimensional vector which can be used as the input for the next layer.

- As the output of a convolution can be N -dimensional, so can the input be an M -dimensional vector. In fact, when processing images, we typically start with three channels. These three channels can then be mapped through convolution to an arbitrary number N of output features (N is often called the depth of the output). The more general convolution function is hence a mapping of an M -dimensional input vector x to an N -dimensional output vector o . For a pixel location (i, j) , we obtain:

$$o_{i,j,n}(x) = \varphi \left(b_n + \sum_{k,l,m} w_{k,l,m,n} \cdot x_{i+k,j+l,m} \right)$$

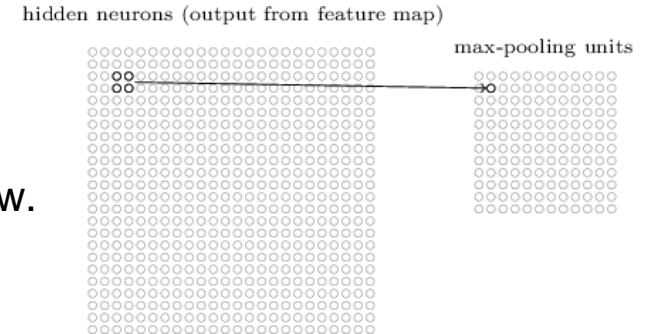
For example, let us assume a 5x5 convolution on three ($M = 3$) input channels, and we want to convolute to $N = 20$ output feature. The above formula contains shared biases b_n for each output feature $1 \leq n \leq N$, and shared weights $w_{k,l,m,n}$ for each of the 5x5 positions of the window, for each channel $1 \leq m \leq M$ and each output feature $1 \leq n \leq N$. Hence, we have 20 biases and $5 \times 5 \times 3 \times 20 = 1500$ weights. The shared parameters are then used for all pixel locations in the image. If we started with a 256x256 input image with 3 channels, the output of the convolution is now a 256x256x20 arrays. Interestingly, we do not need to map the color spaces as the network now can also learn the best linear combination of the channels.



- The special case of a **1x1 convolution** is often used to reduce the dimensionality of the input values. Assume we want to learn a 5x5 convolution with 20 output features and we have 20 input features: we would need to learn $5 \times 5 \times 20 \times 20 = 10'000$ weights and 20 biases (in total 10'020 parameters). A 1x1 convolution can reduce the number of parameters to learn as follows:
 - We can first apply a 1x1 convolution to generate 3 output features (from the 20 input features). We require $1 \times 1 \times 20 \times 3 = 60$ weights and 3 biases for this layer (63 parameters in total).
 - We then feed the 3 features from the 1x1 convolution into a 5x5 convolution with 20 output features. We require $5 \times 5 \times 3 \times 20 = 1'500$ weights and 20 biases (1'520 parameters in total)
 - Overall, the new network structure has 1'583 parameters compared to the 10'020 with the naïve, straightforward mapping.
- An interesting aspect of convolution is that its complexity (number of parameters) is independent of the input size of the network. However, computational complexity (forward and backward steps) depend on the number of input values. For instance, an input sizing for 256x256 is 4 times faster than for a 512x512 sizing. If images are the input, the typical approach is to scale them down to a reasonable size that can be fed into the network. We will see later techniques to deal with scale variance, e.g., recognizing objects at different scales.
- **Strides:** convolution uses a sliding window which is applied at each location to compute an output value. In addition, it is also possible to define how far apart two subsequent windows must lie. A stride of (2,2) means that only every other value in both dimensions is used as the starting location of the window. Thus, only half as many rows and columns are created in the output. Strides can be used to reduce the initial size of the network. A (2,2) stride will lead to 4 times less output neurons. For images, this allows to scale down the size and compute features at various scales.

- Convolution layers are often followed by **Pooling Layers**. Pooling reduces the number of neurons and thus simplify the overall information.

- A pooling layer is again a spatially organized structure. It summarizes the values of a window in the previous layer. For example consider the picture on the right hand side: a 2x2 max-pooling layer outputs the maximum value of the 2x2 window. If we additionally use a stride of (2,2), this reduces the “feature map” by 4 times. If the input consists of multiple channels, then the pooling operator is applied at each channel individually. Here, we do not apply an activation function:

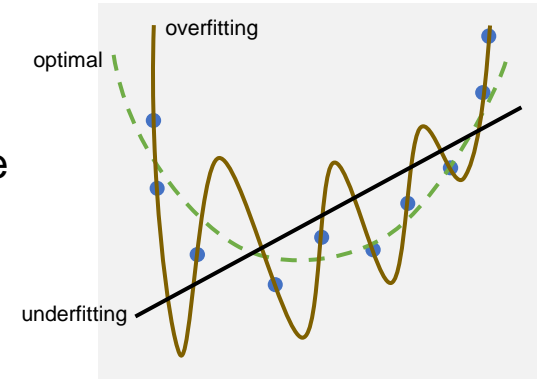


$$o_{i,j,n}(x) = \max_{l,k} x_{i+k,j+l,n}$$

- Next to max pooling, other summarization functions are possible. Typical examples include average pooling and L_2 -Norm pooling.
- In deep learning, for instance image object recognition, pooling layers are an important control mechanism to reduce the spatial size of the representation and with that the number of parameters in the network model. This not only greatly reduces the amount of computation but also reduces the risk of overfitting. Recall that the best model is the simplest one among equally good methods. Also note that pooling only reduce spatial dimensions if the stride is larger than 1. It does, however, not reduce the number of features (depth). For that, a 1x1 convolution is required as described before.

- **Regularization** is an important element in deep learning to prevent overfitting to the training data.

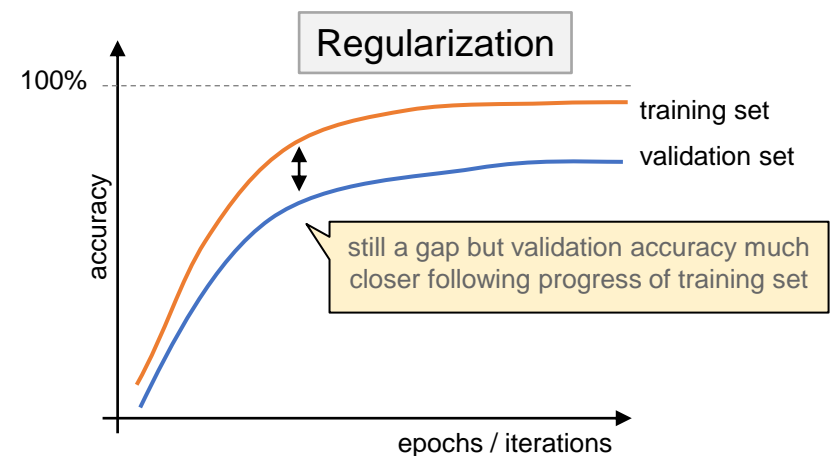
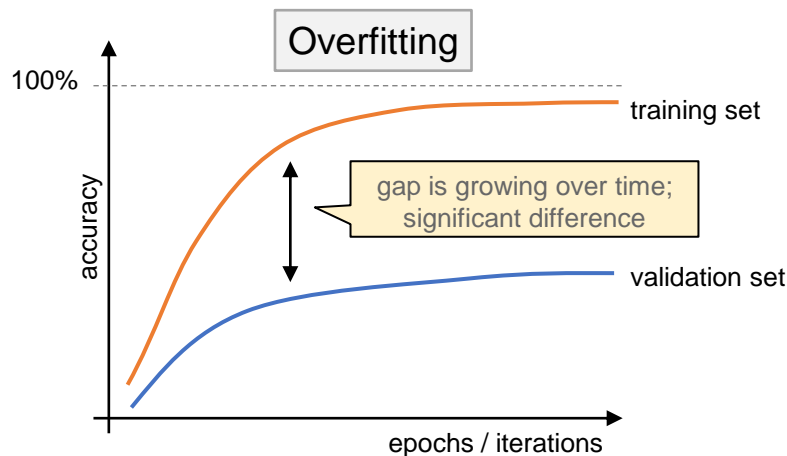
- As we discussed earlier, overfitting occurs if the model has too many parameters and hence memorizes the data rather than generalizing rules from it. The picture on the right hand shows a simple example of what overfitting means. While the models on the right side may use dozens of parameters, a deep neural network can have several millions of parameters. Hence, how do we prevent the network from simply memorizing the input to target mapping, and how can we detect an overfitting problem.



- Overfitting is the lack of generalization and will become evident if we apply a trained to new data items that were not used during training. The validation set can be used to detect overfitting.

Overfitting can be recognized as follows:

- Almost perfect accuracy for the training set at the end of the learning
- Significant lower accuracy for the validation set at the end of the learning
- The gap between training accuracy and validation accuracy is growing over the learning time



- We have several options for regularization
 - **Adjust the network structure** and reduce the number of parameters—not really an option given that we want to learn complex tasks. The success of small networks was rather limited.
 - **Expand the training set**—not always feasible, but we can modify and alter the existing data set. For instance, small rotations, varying brightness, adding noise, Gaussian filters, etc. With a few such modifications, we can create 10 to 100 times more training data without any additional labelling costs.
 - **Adjust the cost function** to prefer simpler models. A simple method is to add a penalty to the cost functions for the use of large weights. Smaller weights (preferably 0) reduce the complexity of the model. This way we can balance overfitting to the training with a penalty for more complex models. Our cost function looks now as follows (L2 regularization):

$$J_{reg}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \frac{\lambda}{2 \cdot |\mathbb{T}|} \sum_i w_i^2$$

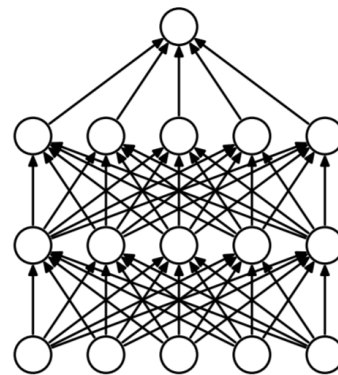
With $|\mathbb{T}|$ being the number of training samples and $\lambda > 0$ the regularization parameter. Note that we only add penalties for the weights but not for the biases. With this, we have a new update for w_i during back propagation. Let Δ_i be the update for w_i without regularization, then:

$$w_i^{(t+1)} = \left(1 - \frac{\eta\lambda}{|\mathbb{T}|}\right) \cdot w_i^{(t)} - \Delta_i$$

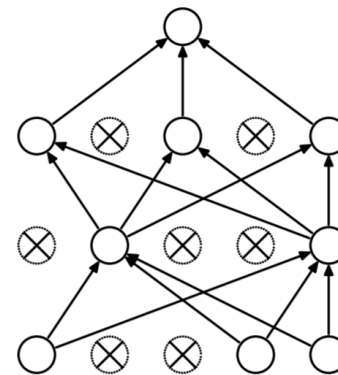
Regularization adds a weight decay factor $\left(1 - \frac{\eta\lambda}{|\mathbb{T}|}\right)$ for each weight, making them gradually smaller unless the gradient compensates enough to increase weights in the learning step. This was shown to greatly reduce the risk of overfitting.

- The **Dropout technique** heuristically adjust the network structure during the learning phase. At any point in time during the learning phase, only parts of the network are active (with a random selection of nodes). This selection can change over time:
 - At each training step, nodes are dropped out with a probability of $1 - p$. Over the learning time, different sets of active nodes learn the training example
 - Feed forward: if a node is dropped out, its output value is set to 0. We keep weights and biases as the node may become active in a subsequent training step
 - Back propagation: if a node is dropped out, it does no longer propagate changes. The weights of connection to/from such a node do not receive an update.
 - The final model for prediction uses all nodes but compensates their weights with $(1 - p)$.

We can interpret the dropout technique as learning many different networks at the same time. Finally, we combine all the individual networks into a single, bigger network. This helped with overfitting as each individual subset of the network has adapted differently to the training set. By “averaging” the networks for prediction, the impact of overfitting in one such sub-network is evened out the other sub-networks (which may have overfitted other aspects of the training set)



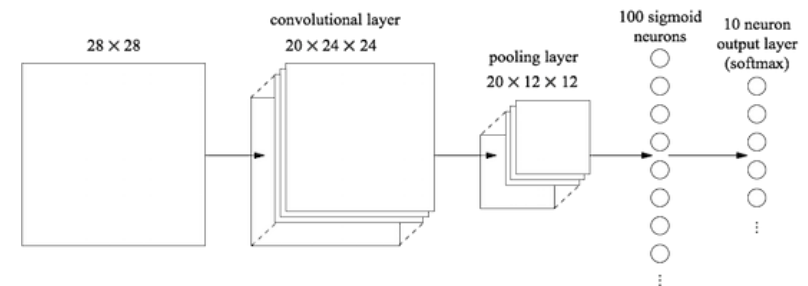
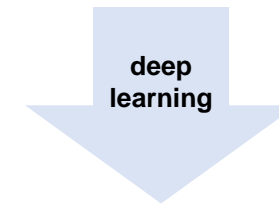
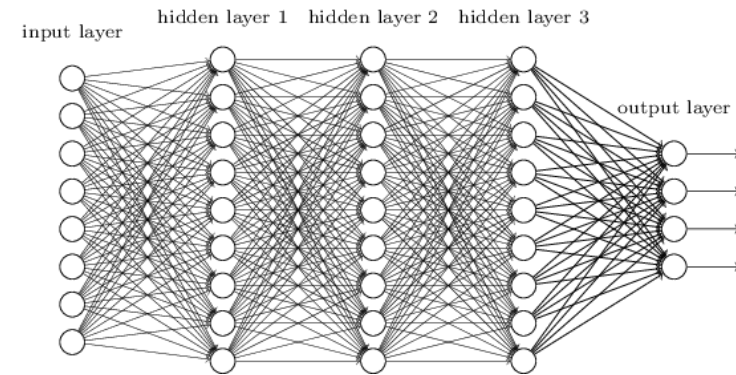
(a) Standard Neural Net



(b) After applying dropout.

- **Putting all together**

- Let us start with a simpler example: the MNIST database (see next page) consists of 28×28 images depicting hand written digits (0, 1, 2, ..., 9)
- The conventional approach with neural network used fully connected hidden layers like in the picture on the top right. Its performance was ok but methods like SVM and k-NN classification proved to be better.
- The deep learning approach: use of convolution and pooling greatly improved performance. The picture on the bottom right show a possible architecture. The first 5×5 convolution produces 20 features with a ReLU activation (here, no padding is applied hence the size of the network reduces to 24×24). A subsequent 2×2 max-pooling layer reduces the spatial dimension to 12×12 (with 20 features). These $12 \times 12 \times 20 = 2880$ elements are fully connected to 100 neurons. Finally, a softmax layer reduces the 100 neurons to 10 classes. The output neuron with the highest value denotes the class for prediction.



- The original black and white images from NIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of anti-aliasing. The images were centered in a 28x28 image by computing the center of mass of the pixels and moving the 20x20 image.
- The data set consists of 60'000 training items and 10'000 test items. The algorithms must learn a prediction method to map an image to one of the 10 classes 0, 1, 2, ..., 9. The error rate is computed against the test data.
- The best method currently (a convolutional network) has an error rate of 0.23%. It is noteworthy to comment that some of the wrongly labelled images are also a challenge for humans to read correctly.
- List of further datasets for machine learning
 - https://en.wikipedia.org/wiki/List_of_datasets_for_machine_learning_research

HANDWRITING SAMPLE FORM

NAME	DATE	CITY	STATE	ZIP
[REDACTED]	8-3-89	MINNEN CITY	Mi	48452

This sample of handwriting is being collected for use in testing computer recognition of hand printed numbers and letters. Please print the following characters in the boxes that appear below:

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9

0123456789	0123456789	0123456789		
87 87	701 701	3752 3752	80759 80759	960941 960941
158 158	4586 4586	32123 32123	832656 832656	82 82
7481 7481	80539 80539	419219 419219	67 67	904 904
61738 61738	729658 729658	75 75	390 390	5716 5716
109334 109334	40 40	625 625	4234 4234	46002 46002

g y x l a k p d s b t z i r u m w f q j e n h o c v

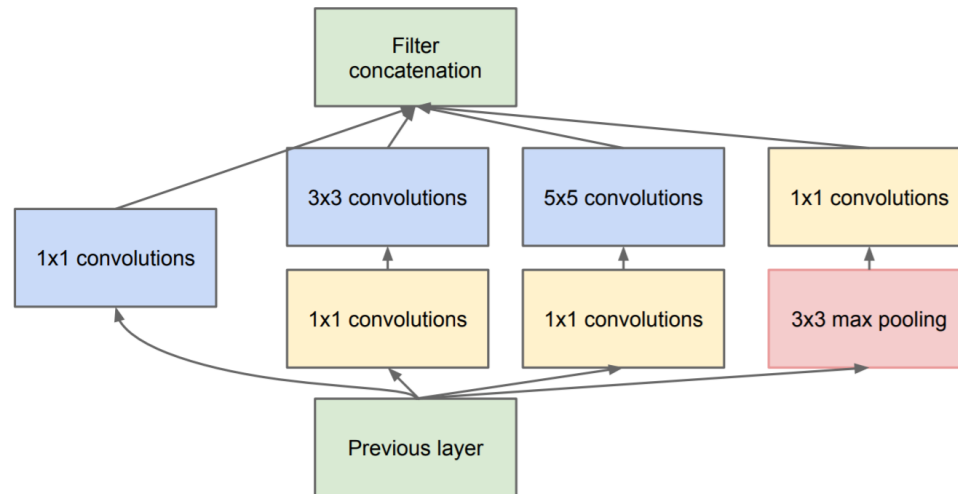
g y x l a k p d s b t z i r u m w f q j e n h o c v
Z X S B N G E C M Y W Q T K F L U O H P I R V D J A
Z X S B N G E C M Y W Q T K F L U O H P I R V D J A

Please print the following text in the box below:
We, the People of the United States, in order to form a more perfect Union, establish Justice, insure domestic Tranquility, provide for the common Defense, promote the general Welfare, and secure the Blessings of Liberty to ourselves and our posterity, do ordain and establish this CONSTITUTION for the United States of America

We, the People of the United States, in order to form a more perfect Union, establish Justice, insure domestic Tranquility, provide for the common Defense, promote the general Welfare, and secure the Blessings of Liberty to ourselves and our posterity, do ordain and establish this CONSTITUTION for the United States of America.

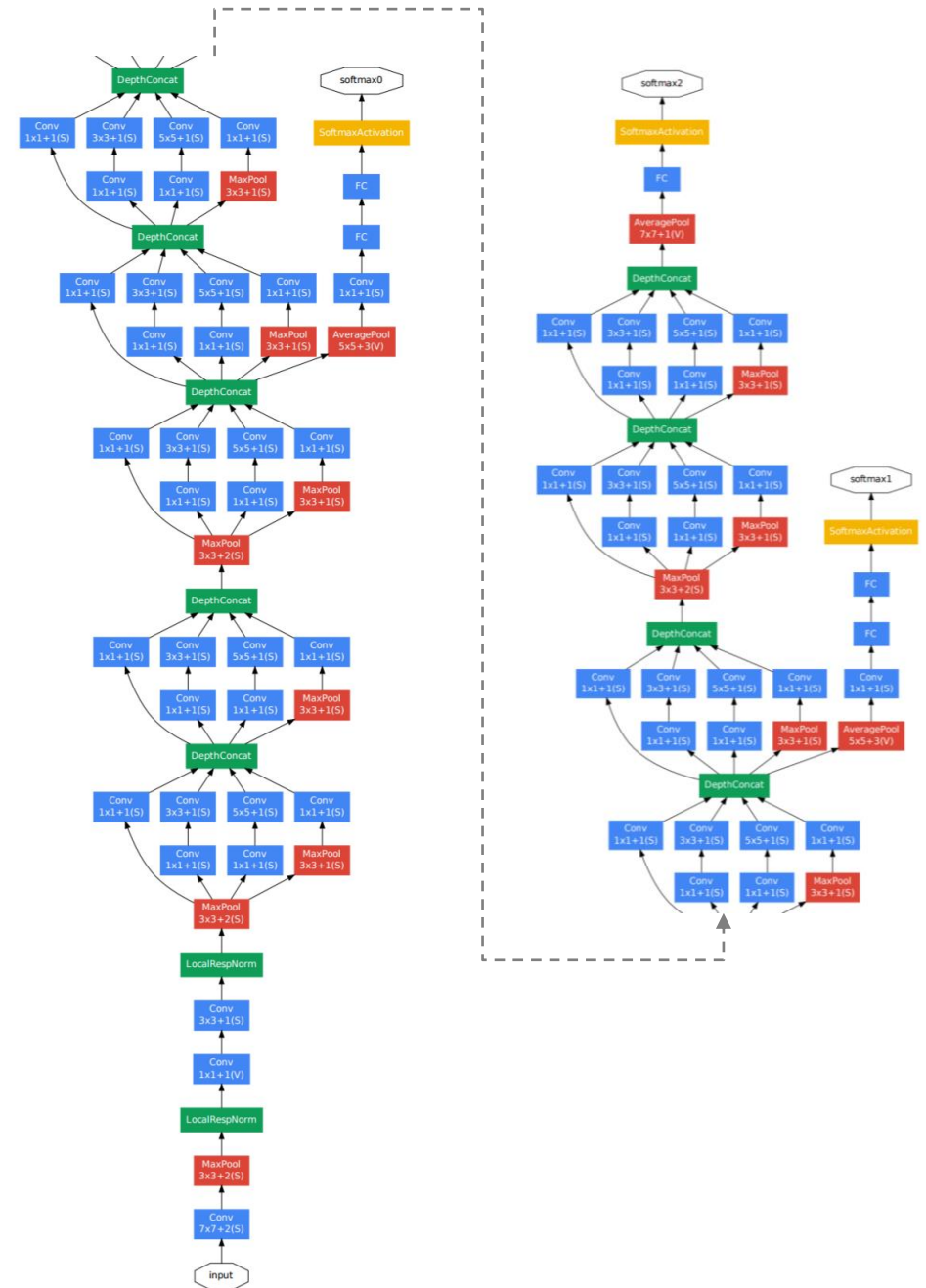
- **GoogleLeNet for image classification**

- GoolgLeNet was the winner of the ILSVRC 2014 Classification Challenge. The contest consisted of 500k images with object labeling in 200 classes.
- A key ingredient of their network architecture included the use of inception modules which are building blocks for the network as shown below:
 - The inception module applies different operators on the output of a previous layer. In the example below, 1x1, 3x3, 5x5 convolutions and a 3x3 max pooling are all applied in parallel. Their output is then concatenated to produce the output features. The idea is that the network should learn itself, which of the operator works best for certain scenarios.
 - To control the complexity of the model, 1x1 convolutions (marked in yellow) are added to reduce the number of features. As previously discussed, this greatly helps to reduce the computational complexity of a 3x3 or 5x5 convolution.



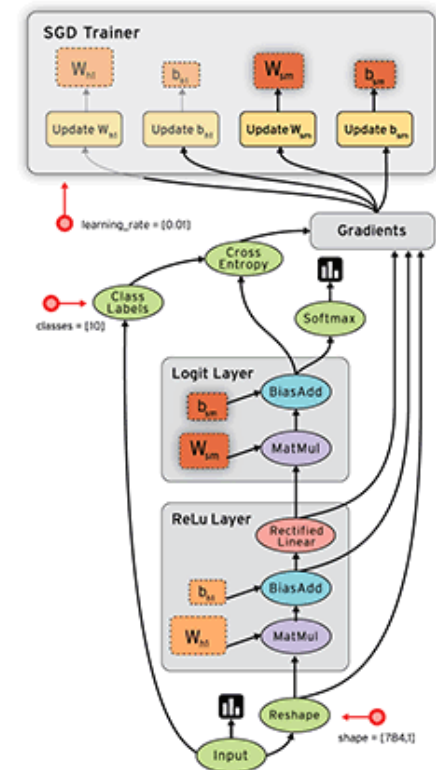
- The full architecture of GoogleLeNet for image classification
 - Input: 224x224 RGB images

Type	size/stride	output	#params	#ops
convolution	7×7/2	112×112×64	2.7K	34M
max pool	3×3/2	56×56×64		
convolution	3×3/1	56×56×192	112K	360M
max pool	3×3/2	28×28×192		
inception (3a)		28×28×256	159K	128M
inception (3b)		28×28×480	380K	304M
max pool	3×3/2	14×14×480		
inception (4a)		14×14×512	364K	73M
inception (4b)		14×14×512	437K	88M
inception (4c)		14×14×512	463K	100M
inception (4d)		14×14×528	580K	119M
inception (4e)		14×14×832	840K	170M
max pool	3×3/2	7×7×832		
inception (5a)		7×7×832	1072K	54M
inception (5b)		7×7×1024	1388K	71M
avg pool	7×7/1	1×1×1024		
dropout -40%		1×1×1024		
linear		1×1×1000	1000K	1M
softmax		1×1×1000		



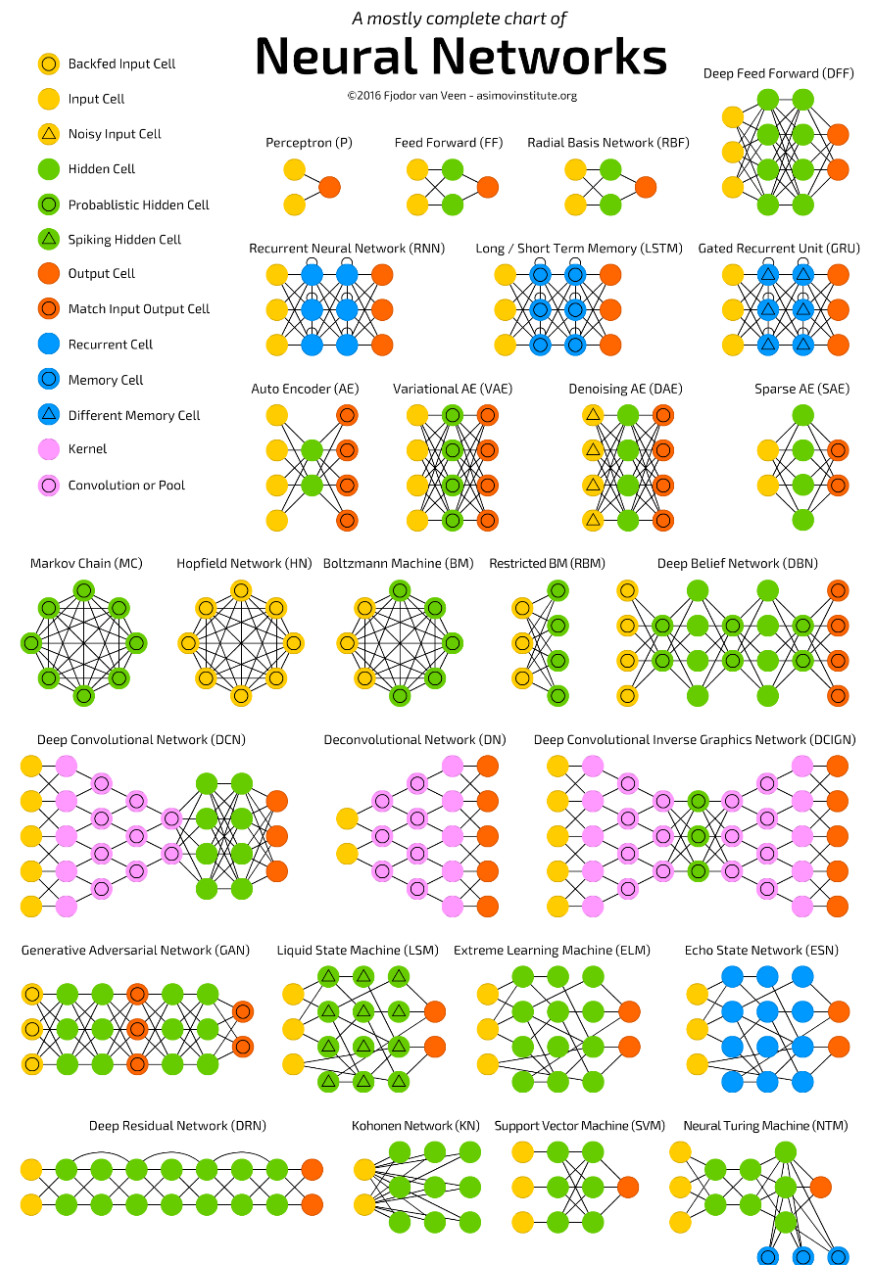
- **Tensorflow**

- Tensorflow was developed by the Google Brain team, initially for Google internal use only. But meanwhile the framework is openly available under Apache 2.0 license and provides a simple to use Python programming front end to its core.
- The term tensor stands for an arbitrary dimensional array holding the data values (often float32).
- Tensorflow has two elements
 - Nodes are operators on input tensors and produce an output tensor
 - Data edges combine nodes and connect outputs with inputs
- The Python front-end provides a simple way of building these graphs based on constants, variables and a rich set of defined operators. In the context of deep learning, most known methods have been implemented into tensorflow allowing for an efficient way of learning and applying a network
- Another aspect of tensorflow is the distributed execution of the graph and the support for CUDA (GPU based operations) and parallel execution of operations. The largest networks can span hundreds of machines and can run against thousands of CUDA cores accelerating computations of large graphs. All this is transparent to the end-user, i.e., the user only must define the graph and tensorflow considers the fastest way to compute the graph.
- For more information see: www.tensorflow.org



- In this chapter, we only looked at deep learning for spatial data sets (images, videos). But there is a great number of further architecture extensions to support, for instance, natural language processing, memorization of facts and data, and so on.
- The Asimov Institute published in 2016 a map outlining the neural network zoo

<http://www.asimovinstitute.org/neural-network-zoo/>



4.11 Literature and Links

- Image Features
 - Wikipedia, List of color spaces and their uses, retrieved 2017 from https://en.wikipedia.org/wiki/List_of_color_spaces_and_their_uses
 - Y. Rui, T. Huang, and S.F. Chang, **ContentBased Image Retrieval: A Survey**, Journal of Visual Communication and Image Representation, Academic Press, March 1999.
 - M. Flickner, H. S. Sawhney, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker, **Query by Image and Video Content: The QBIC System**. IEEE Computer, 28(9):23-32, 1995.
 - M. Stricker and A. Dimai. **Color Indexing with Weak Spatial Constraints**. In Storage and Retrieval for Image and Video Databases (SPIE), volume 2670 of SPIE Proceedings, San Diego/La Jolla, CA, USA, Feb. 1996.
 - B.S. Manjunath and W.Y. Ma, **Texture Features for Browsing and Retrieval of Image Data**, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 18:8, p. 837-842, 1996.
 - A. Vailaya, **Shape-Based Image Retrieval**, Master Thesis, Michigan State University, 1996.
 - S. Berchtold, D. A. Keim, and H.-P. Kriegel. **Section Coding: Ein Verfahren zur Ähnlichkeitssuche in CAD-Datenbanken**. In Datenbanksysteme in Büro, Technik und Wissenschaft, Ulm, Germany, Mar. 1997. Springer.
 - A. Chalechale and A. Mertins. **Angular Radial Partitioning for Edge Image Description**. In Proc. 7th International Symposium on DSP for Communication Systems (DSPCS03), Coolangatta, Australia, Dec. 2003.
 - Navneet Dalal and Bill Triggs, **Histograms of Oriented Gradients for Human Detection**, <http://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf>
 - Lowe, David G. (1999). "[Object recognition from local scale-invariant features](#)". Proceedings of the International Conference on Computer Vision. 2. pp. 1150–1157. doi:10.1109/ICCV.1999.790410
 - R. Szeliski, **Computer Vision: Algorithms and Applications**, Textbook, <http://szeliski.org/Book/>

- Frameworks and Libraries
 - **OpenCV** (<https://opencv.org>) is an advanced computer vision library original written for C/C++. But there are also bindings for Python, Java, and other languages.
 - **scikit-image** (<http://scikit-image.org>) is an advanced computer vision library written in Python. It provides all basic image manipulation operations as well as advanced feature extraction algorithms (however, not SIFT but alternative approaches to SIFT)
 - **Librosa** (<http://librosa.github.io/librosa/>) is a Python library for advances audi and music analysis. It provides base algorithms to create music retrieval systems.
- Interesting courses at other universities
 - Multimedia Content Analysis, National Chung Cheng University, Taiwan, https://www.cs.ccu.edu.tw/~wtchu/courses/2014f_MCA/lectures.html#00
 - Computer Vision, University of Washington, USA, <https://courses.cs.washington.edu/courses/cse455/>
 - Computer Vision, Penn State University, USA, <http://www.cse.psu.edu/~rtc12/CSE486/>
 - Computer Vision, University of Illinois, USA, <https://courses.engr.illinois.edu/cs543/sp2012/>
 - Computational Photography, University of Illinois, USA, <https://courses.engr.illinois.edu/cs498dh/fa2011/>

- Machine Learning
 - Quinlan, J. R. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.
 - C. Szegedy et al, [Going Deeper with Convolutions](#), *Computer Vision and Pattern Recognition (CVPR)*, 2015.
 - Mitchell, Tom M. *Machine Learning*. McGraw-Hill, 1997.
 - M. Nielsen, [Neural Networks and Deep Learning](#), free online book, Dec 2017.
 - I. Goodfellow, *Deep Learning* (Adaptive Computation and Machine Learning series), 2016. Free online version available at: <http://www.deeplearningbook.org>
 - Tensorflow, Apache 2.0, <https://www.tensorflow.org/>
 - Scikit-learn, BSD, <http://scikit-learn.org/>
 - Online Neural Network: <http://playground.tensorflow.org/>